

Einführung in die Programmiersprache C

Begleitheft zum Kurs
für
Mathematisch–technische
Assistentinnen und Assistenten

gehalten am



UNIX
Begleitheft zum Kurs für
Mathematisch-technische
Assistenten und Assistenten

Rechenzentrum
RWTH Aachen
Seffenter Weg 23
D-52074 Aachen
Tel. (0241) 80-4900

Verantwortlich für den Inhalt:
Dr. Wilhelm Hanrath

erstellt und bearbeitet von:
Dr. Wilhelm Hanrath

(Der Text wurde mit $\text{\LaTeX}$ erstellt)

Inhaltsverzeichnis

Vorwort	vii
1 Einleitung	1
1.1 Allgemeine Grundlagen	1
1.1.1 Modell eines Rechners:	1
1.1.2 Übersicht über “unsere Rechnerlandschaft”:	1
1.1.3 Kompletter Name eines Rechners (Internetname):	1
1.1.4 Aufgabe der Praktikumsrechner	1
1.1.5 Aufgabe des File-Servers	2
1.1.6 Aufgabe des Mail-Servers	2
1.1.7 Weitere Rechner im Netz mit Spezialaufgaben	3
1.1.8 Die SUN-Tastatur	3
1.2 Grundlagen zu UNIX	5
1.2.1 An- und Abmelden	5
1.2.2 Die Shell als Kommandointerpreter	6
1.2.3 Passwort ändern	8
1.2.4 Das UNIX-Dateisystem	8
1.2.5 Einfache Dateiorientierte Kommandos	10
1.2.6 Ein- und Ausgabeumlenkung	15
1.2.7 Metazeichen, Cut & Paste	17
1.2.8 Drucken von Dateien	18
1.3 Nutzung des Internets	20
1.3.1 Informationen aus dem Internet	20
1.3.2 Elektronische Post, E-Mail	21
1.4 Programmerstellung	22
1.4.1 Prinzipielles zur Programmerstellung	22
1.4.2 C-Programm-Erstellung unter UNIX	22
1.4.3 Geschichtliche Entwicklung von C	23
1.5 Der vi-Editor	24
1.5.1 Aufruf des vi-Editors	24
1.5.2 Die Modi des vi	24
1.5.3 Löschen von Text	27
1.5.4 Suchen, Suchen und Ersetzen	28
1.5.5 Pufferoperationen	30
1.5.6 Sonstige nützliche vi-Befehle	31
2 Grundlagen zur C-Programmierung	39
2.1 Einfachste C-Programme	39
2.2 Erlaubte Zeichen, Schlüsselworte, Operatoren	45
2.2.1 Zeichen in C-Quelltexten	45
2.2.2 Schlüsselwörter	45
2.2.3 C-Operatoren	46

3	Elementare Datentypen und deren Behandlung	48
3.1	Ganzzahlige Datentypen	48
3.1.1	Ganzzahlige Konstanten	52
3.1.2	Definition und Initialisierung ganzzahliger Variablen	52
3.1.3	Operationen mit ganzzahligen Werten	53
3.1.4	Ausgabe ganzzahliger Werte	55
3.1.5	Eingabe ganzzahliger Werte	57
3.1.6	Beispielprogramme	61
3.2	Gleitpunkttypen	61
3.2.1	Gleitpunktkonstanten	63
3.2.2	Definition von Gleitpunktvariablen	64
3.2.3	Operationen mit Gleitpunktwerten	64
3.2.4	Ausgabe von Gleitpunktwerten	65
3.2.5	Eingabe von Gleitpunktwerten	66
3.3	Zeichen	67
3.3.1	Zeichenkonstanten	69
3.3.2	Definition und Initialisierung von Variablen vom Typ <code>char</code> . .	70
3.3.3	Operationen mit <code>char</code> -Werten	70
3.3.4	Ein-/Ausgabe von Zeichen	71
4	Ablaufkontrolle	73
4.1	Auswahanweisungen	73
4.1.1	Die einfache <code>if</code> -Anweisung	73
4.1.2	Die <code>if-else</code> -Anweisung	75
4.1.3	Geschachtelte <code>if</code> -Anweisungen	76
4.2	Zählschleifen	77
4.3	Die <code>while</code> -Schleife	80
4.4	Die <code>do-while</code> -Schleife	81
4.5	Die <code>switch</code> -Anweisung	81
4.6	Die <code>goto</code> -Anweisung	84
4.7	Weitere Beispiele zu Kontrollstrukturen	84
5	Felder	87
5.1	Grundlagen zu Feldern	87
5.1.1	Explizite Initialisierung von Feldern	89
5.2	Zeichenfelder	90
5.2.1	Ausgabe von Strings	92
5.2.2	Eingabe von Strings	93
5.2.3	Weitere Bibliotheksfunktionen zur Stringbearbeitung	94
5.3	Mehrdimensionale Felder	95
5.3.1	Initialisierung mehrdimensionaler Felder	97

6	Funktionen	99
6.1	Grundlagen zu Funktionen	99
6.2	Funktionsargumente, Funktionsparameter, <i>Call by Value</i>	104
6.3	Funktionen ohne Argumente	106
6.4	Ergebnistypen	107
6.5	Verwendung mehrerer Funktionen, Rekursion	108
6.6	Wertverändernde Funktionen, Adress-Variablen	111
6.6.1	Der Adress-Operator	114
6.6.2	Adress-Variablen	114
6.6.3	Der Verweis-Operator	115
6.6.4	Wertverändernde Funktionen	116
6.6.5	Typlose Adressen	118
6.6.6	Felder und Adressen von Adress-Variablen	119
6.7	Felder und Funktionen	119
6.7.1	Eindimensionale Felder als Funktionsargumente	119
6.7.2	Funktionsergebnisse vom Adresstyp	123
6.7.3	Mehrdimensionale Felder als Funktionsargumente	124
6.7.4	Alternative zu mehrdimensionalen Feldern als Funktionsargumente	125
6.8	Mehrere Quelltexte	126
6.8.1	Compilieren von aus mehreren Quelltexten bestehenden Programmen	127
6.8.2	Eigene Header-Dateien	129
7	Verschiedenes	130
7.1	Argumente und Funktionsergebnis von <code>main</code>	130
7.1.1	Funktionsergebnis von <code>main</code>	130
7.1.2	Argumente der Kommandozeile	130
7.2	Aufzählungstypen	132
7.3	Selbstdefinierte Typen	133
7.4	Das Schlüsselwort <code>const</code>	134
7.4.1	Adress-Variablen und <code>const</code>	135
7.4.2	Zeiger auf <code>const</code> als Funktionsparameter	136
7.5	Das Schlüsselwort <code>volatile</code>	137
8	Operatoren, Ausdrücke und Anweisungen	138
8.1	Operatoren	138
8.2	Ausdrücke	144
8.3	Typumwandlungen	145
8.3.1	Ganzzahlige Aufwertung	145
8.3.2	Wie wird umgewandelt?	145
8.3.3	Wann wird umgewandelt?	146
8.4	Anweisungen	146

9	Speicherklassen	148
9.1	Speicherklassen von Variablen	148
9.1.1	Automatische Variablen	148
9.1.2	Register-Variablen	151
9.1.3	Statische interne Variablen	152
9.1.4	Externe Variablen	154
9.1.5	Statische externe Variablen	156
9.2	Speicherklassen von Funktionen	158
9.2.1	Statische Funktionen	158
10	Der Präprozessor	159
10.1	Einfügen von Dateien	159
10.2	Ersetzen von Makros	159
10.3	Bedingte Übersetzung	162
10.4	Vordefinierte Makros	163
10.5	Weitere Präprozessoranweisungen	164
11	Die Standardbibliothek	166
11.1	Standardein-/ausgabe	166
11.1.1	Elementare Ein-/Ausgabe	167
11.1.2	Dateizugriff	171
11.1.3	Weitere Dateioperationen	174
11.1.4	Fehlerbehandlung bei Datenströmen	175
11.1.5	Direkte Ein-/Ausgabe, Dateipositionierung	176
11.2	Universelle Hilfsfunktionen	177
11.3	Variable Argumentliste	180
11.4	Mathematische Funktionen	181
11.5	Tests für Zeichen	183
11.6	Funktionen für Zeichenketten	184
11.7	Fehlersuche	186
11.8	Globale Sprünge	186
11.9	Signale	187
11.10	Datum und Uhrzeit	188
11.11	Systemabhängige Größen und Typen	190
12	Strukturen	192
12.1	Einleitung	192
12.2	Adress-Variablen vom Strukturtyp	195
12.3	Funktionsargumente und -ergebnisse vom Strukturtyp	196
12.3.1	Strukturen als Funktionsargumente	196
12.3.2	Strukturen als Funktionsergebnisse	197
12.4	Struktur-Komponenten vom Strukturtyp	199
12.5	Speicherlücken, Alignment	200
12.6	Selbstreferierende Strukturen	201

12.7 Unions	201
12.8 Bitfelder	202
13 Mehr zu Adressen	204
13.1 Adress-Arithmetik	204
13.2 Adressen von Funktionen	206
Literaturverzeichnis	209
Index	210

Abbildungsverzeichnis

1 Modell eines Rechners	34
2 Unsere Rechnerlandschaft	35
3 Internetname	35
4 SUN-Tastatur	36
5 UNIX-Dateibaum	36
6 Ausgabe des <code>ls -l</code> -Kommandos	37
7 Erläuterung zum Rechtevektor	37
8 Programmerstellung	38
9 Modi des <code>vi</code> -Editors	38
10 Programmaufbau im Arbeitsspeicher	39
11 Prinzipieller Aufbau eines C-Programms	39
12 Programmfluss	41
13 Funktionsaufruf	42
14 Speichergrößen	48
15 Eingabefließband	58
16 Dezimaler, oktaler und hexadezimaler Wert eines Zeichens	67
17 Struktogrammelement: <code>if-else</code> -Anweisung	76
18 Struktogrammelement: Fallunterscheidung zwischen mehreren Fällen	77
19 Struktogrammelement: Zählschleife	79
20 Struktogrammelement: kopfgesteuerte Schleife	80
21 Struktogrammelement: fußgesteuerte Schleife	81
22 Struktogrammelement: <code>switch</code> -Anweisung	83
23 Datenfluss beim Funktionsaufruf	105
24 Aufrufbarkeit einer Funktion	109
25 Aufruf einer rekursiven Funktion	111
26 Speicherlücken	201

Tabellenverzeichnis

1 Steuertasten	4
--------------------------	---

2	Schlüsselworte	46
3	C-Operatoren	46
4	Zeichensatztablelle	68
5	ASCII-Steuerzeichen	69
6	C-Operatoren	138

Vorwort

Dies ist die (provisorische) Ausarbeitung des C-Kurses für Mathematisch-technische Assistentinnen und Assistenten, den ich seit Herbst 1995 im Rechenzentrum der RWTH-Aachen in dieser (oder ähnlicher) Form abhalte.

Dieser Kursus findet zu Beginn des dreijährigen Ausbildungsganges statt, so dass nicht von einheitlichen Vorkenntnissen der Auszubildenden — meist Abiturientinnen und Abiturienten unmittelbar nach ihrer Schulzeit — im Datenverarbeitungsbereich und insbesondere im Umgang mit UNIX-Rechnern ausgegangen werden kann.

Aus diesem Grund wird zunächst ein wenig auf den Umgang mit Rechnern im Allgemeinen und insbesondere auf den Umgang mit unseren Praktikumsrechnern eingegangen.

Ziel des eigentlichen Kurses ist dann, einem Programmierneuling die Grundlagen der Programmierung in C möglichst behutsam näher zu bringen.

Aachen, im August 1998

Wilhelm Hanrath

1 Einleitung

1.1 Allgemeine Grundlagen

1.1.1 Modell eines Rechners:

1.1.2 Übersicht über “unsere Rechnerlandschaft“:

In folgendem Schaubild ist schematisch der Aufbau unseres Rechnernetzes dargestellt:

Die Menge aller weltweit verbundener Rechner heißt **Internet**. Somit gehören auch unser File-Server und unsere Praktikumsrechner zum Internet.

1.1.3 Kompletter Name eines Rechners (Internetname):

Der symbolische Internetname eines Rechners hat mehrere, durch Punkte getrennte Bestandteile, aus denen die Einbindung des Rechners ins Internet abgelesen werden kann:

Die Länderkennung (etwa **.de** für Deutschland, **.uk** für England, **.fr** für Frankreich) wird, vor allem in Amerika, nicht durchgängig verwendet. An deren Stelle wird bisweilen auch eine Kennung benutzt, aus der man das Einsatzgebiet der entsprechenden Institution ablesen kann, etwa **.edu** für Schulen jeder Art, **.gov** für Behörden, **.com** für gewerblich orientierte Unternehmen.

1.1.4 Aufgabe der Praktikumsrechner

Die Aufgabe der Praktikumsrechner ist: Ausführen von Kommandos und Programmen.

Der Praktikumssteilnehmer sitzt vor Bildschirm und Tastatur des Rechners, gibt Kommandos und Befehle ein; der Rechner führt diese Befehle aus und gibt dabei Meldungen und Ergebnisse auf dem Bildschirm aus.

Jeder Praktikumsrechner selbst ist komplett ausgestattet, verfügt über ein Betriebssystem und kann im Prinzip selbständig arbeiten.

Aus organisatorischen Gründen liegen die benutzereigenen Daten und Programme jedoch nicht lokal auf dem Praktikumsrechner, sondern zentral auf dem File-Server, damit jeder Praktikumssteilnehmer an jedem Praktikumsrechner arbeiten kann und “auf jedem“ seine eigenen Daten vorfindet.

Unsere Praktikumsrechner sind SUN-Workstations mit 128 MByte Hauptspeicher und 1 GByte Festplatte und laufen (zur Zeit) unter dem Betriebssystem Solaris 2.5.1 (UNIX-Derivat von SUN).

1.1.5 Aufgabe des File-Servers

Unser File-Server ist eine SGI-Workstation, verfügt über einen Hauptspeicher von 64 MByte und hat eine Festplattenkapazität von etwa 100 GByte. Er verfügt über zwei Prozessoren und über einige, besonders schnelle Netzanbindungen. Das Betriebssystem ist ebenfalls UNIX.

Die Aufgaben des File-Servers sind:

- Bereitstellen von weiteren Programmen und Daten, etwa Texteditoren, Compiler, Graphik-Programme. Diese sind nur einmal auf dem File-Server abgespeichert und jeder Benutzer eines Praktikumsrechners kann auf diese Programme zugreifen. (Diese Programme brauchen somit nicht auf jedem Praktikumsrechner selbst vorhanden zu sein!)

Insbesondere wird das Einspielen neuer Software vereinfacht!

- Benutzerverwaltung:

Der File-Server hält für jeden Benutzer (Praktikumsteilnehmer) einen Speicherbereich bereit. Der Benutzer kann dort seine persönlichen Daten und Programme ablegen und diese sind vor dem Zugriff anderer Benutzer geschützt. Desweiteren ist es unerheblich, an welchem der Praktikumsrechner der Benutzer gerade arbeitet — er findet immer seine Original-Daten vor!

Um den Schutz der persönlichen Daten zu gewährleisten, muß sich jeder Benutzer mit einer persönlichen Kennung (Benutzername) und einem geheimen Passwort beim Praktikumsrechner anmelden und nach Beendigung seiner Arbeit sich auch wieder abmelden! Auch die Information über bekannte Benutzer und deren Passworte wird zentral verwaltet, so dass nicht jeder Benutzer auf jedem Praktikumsrechner ein eigenes Passwort haben muss! (Ändert ein Benutzer sein Passwort auf einem der Rechner, so wird es automatisch sofort für alle Rechner abgeändert!)

- Einfachere Datensicherung (Backup) ermöglichen.

In regelmäßigen Zeitabständen wird der Festplatteninhalt des Servers auf Magnetkassetten o.ä. (Tertiärspeicher) kopiert und diese Kopien werden aufbewahrt.

- ...

1.1.6 Aufgabe des Mail-Servers

Die Aufgabe unseres Mail-Servers `postc1` ist es, E-Mail (elektronische Post) u.a. für die Praktikumssteilnehmer zwischenzuspeichern.

Jeder Praktikumssteilnehmer hat eine weltweit erreichbare E-Mail-Adresse:

`benutzerkennung@postc1.rz.rwth-aachen.de`

An eine solche Adresse verschickte Post kommt auf diesem Rechner `postc1` an und wird dort gespeichert.

Jeder Praktikumssteilnehmer kann vom Praktikumsrechner aus nachfragen, ob E-Mail für ihn angekommen ist und kann ggf. die Post vom Rechner `postc1` in seinen persönlichen Arbeitsbereich auf den File-Server kopieren! (Einzelheiten werden später

erläutert!)

Der Rechner `postc1` ist i. Allg. ständig eingeschaltet, so dass, selbst wenn die Praktikumsrechner ausgeschaltet sind und kein Praktikumssteilnehmer aktiv an einem Rechner arbeitet, Post ständig entgegengenommen wird.

Der Rechner `postc1` ist ein PC, welcher (zur Zeit) unter dem Betriebssystem LINUX, dem frei verfügbaren UNIX für PC's, läuft.

1.1.7 Weitere Rechner im Netz mit Spezialaufgaben

Im Rechnernetz des Rechenzentrums gibt es eine Reihe von weiteren Rechnern mit Spezialaufgaben, deren Dienstleistungen von jedem Praktikumssteilnehmer implizit in Anspruch genommen werden. Dies sind u.a.:

- **Nameserver:**

Dies sind Rechner, welche die Internetnamen (etwa: `sunc1.rz.rwth-aachen.de`) in zahlwertige Netzadressen, *IP-Adresse* genannt, (etwa 137.226.70.101) umwandeln. (Natürlich funktioniert die Kommunikation auf dem Netz nur mittels Zahlen!)

- **Gateways:**

Das sind Rechner, welche verschiedene Netze aneinander anbinden (etwa das lokale Netz einer Abteilung an das Netz des Rechenzentrums — dieses an das der Hochschule, welches wiederum an das Netz der Telekom (o.ä.) angebunden ist).

- **NIS-Server:**

Speichert u.a. Benutzerinformationen, so dass diese nicht auf jedem Rechner lokal vorhanden sein muss. Alle Rechner eines Bereiches fragen derartige Informationen beim NIS-Server ab.

- **WWW-Server:**

Rechner, welche beliebige, weltweit abrufbare Informationen eines Unternehmens, einer öffentlichen Einrichtung oder einer Privatperson bereitstellen.

- ...

1.1.8 Die SUN-Tastatur

Die Sun-Tastatur ist in folgendem Schaubild schematisch dargestellt:

In folgenden Erläuterungen sind die Zeilen von unten nach oben von 1 bis 6 durchnumeriert und pro Zeile die einzelnen Tasten von links nach rechts. Die Taste (8, 3) ist somit die 8.-te Taste (von links) der 3.-ten Zeile (von unten) — also das G.

Einige Sondertasten (die in der vierten Spalte mit $\times$ gekennzeichneten Tasten sind gleichzeitig mit anderen Tasten zu betätigen).

Tasten haben i. Allg. zwei Bedeutungen (meist Groß- und Kleinbuchstaben). Durch einfaches Drücken einer solchen Taste erhält man den entsprechenden Kleinbuchstaben bzw. das auf der Taste unten stehende Zeichen. Durch gleichzeitiges Drücken der *Shift-Taste* erhält man den entsprechenden Großbuchstaben bzw. das auf der Taste oben stehende Zeichen..

Bezeichnung	Abkürzung	Position	
<i>Control-Taste</i>	<ctrl>	(3, 3)	×
<i>Escape-Taste</i>	<esc>	(3, 5)	
<i>Return-Taste</i>	<cr>	(15, 3)	
<i>Tabulator-Taste</i>	<tab>	(3, 4)	
<i>Caps-Lock-Taste</i>		(3, 1)	
<i>Backspace</i>	<bs>	(16, 4)	
<i>Leer-Taste</i>		(6, 1)	
<i>Shift-Taste</i>		(3, 2) oder (14, 2)	×
<i>Num-Lock-Taste</i>		(21, 5)	
<i>Delete-Taste</i>		(17, 4)	

Tabelle 1: Steuertasten

Die *Caps-Lock-Taste* und die *Num-Lock-Taste* ändern den “Zustand“ der Tastatur. Wird die *Caps-Lock-Taste* einmal gedrückt, so wird fortan jeder eingetippte Buchstabe groß geschrieben (analog zum gleichzeitigen Drücken der *Shift-Taste*). Dies betrifft jedoch nur die Buchstaben, die übrigen Tasten behalten ihre zwei Bedeutungen — die gewöhnliche und bei gleichzeitigem Drücken der *Shift-Taste* die zweite. Dieser *Caps-Lock-Zustand* wird durch ein grünes Licht in der *Caps-Lock-Taste* angezeigt. Nochmaliges Drücken der *Caps-Lock-Taste* hebt diesen Zustand wieder auf (und das Licht erlöscht wieder).

Die normale Bedeutung der Tasten des *Numerik-Blockes* (jeweils die letzten 3–4 Tasten der Zeilen 1 bis 5) ist die jeweils unten stehende. Durch einmaliges Drücken der *Num-Lock-Taste* erhalten alle diese Tasten die jeweils oben stehende Bedeutung. Dieser Zustand wird durch ein Licht in der *Num-Lock-Taste* angezeigt und er wird durch nochmaliges Drücken der *Num-Lock-Taste* wieder aufgehoben.

Auf die Bedeutung der anderen Tasten wird hier nicht weiter eingegangen.

1.2 Grundlagen zu UNIX

1.2.1 An- und Abmelden

Die Praktikumsrechner sind i. Allg. immer eingeschaltet und **dürfen von den Praktikumsteilnehmern niemals ausgeschaltet werden!**

U. U. ist jedoch der Monitor noch nicht eingeschaltet — dieser kann durch die Praktikumssteilnehmer eingeschaltet und nach der Sitzung auch wieder ausgeschaltet werden (Schalter vorne rechts unterhalb des Bildschirms).

Bei eingeschaltetem Monitor leuchtet neben oder über dem Einschalter eine grüne Kontrolllampe.

Auch bei eingeschaltetem Monitor kann der Bildschirm dunkel sein — in diesem Fall ist die *Return-Taste* (im Folgenden durch `<cr>` abgekürzt) zu betätigen.

Unsere Praktikumsrechner laufen unter der graphischen Bedienoberfläche *CDE* (*Common Desktop Environment*), d.h. auch der Anmeldevorgang läuft unter Kontrolle dieser graphischen Oberfläche ab.

Zunächst erscheint mitten auf dem Bildschirm ein Fenster, in dem nach einer Benutzererkennung gefragt wird. Der Mauszeiger ist auf das Eingabefeld in diesem Fenster zu positionieren und das Eingabefeld ist durch Drücken der linken Maustaste zu aktivieren. Anschließend kann über die Tastatur eine gültige Benutzererkennung eingegeben und mittels `<cr>` abgeschickt werden (Identifizierung).

Anschließend erscheint mitten auf dem Bildschirm ein neues, dem ersten ähnliches, Fenster, in dem nach dem zur Benutzererkennung gehörenden Passwort gefragt wird. Hier ist analog zur Benutzererkennung im Eingabefeld das Passwort einzugeben und durch `<cr>` abzuschicken (Authentifizierung). Im Gegensatz zur Benutzererkennung wird das eingegebene Passwort beim Schreiben nicht auf dem Bildschirm ausgegeben, so dass man dieses “blind“ eintippen muss.

Benutzerkennungen und zugehörige, vorläufige Passworte werden im Praktikum ausgeteilt.

Bei erfolgloser Anmeldung (etwa falsches Passwort oder bei der Passworteingabe vertippt) erscheint ein Fenster mit einer entsprechenden Fehlermeldung. In diesem Fenster befindet sich ein Knopf (*Button*) mit der Aufschrift **ok**. Dieser Knopf ist zu betätigen, d.h. der Mauszeiger muss auf diesen Knopf positioniert werden und dann muss die linke Maustaste gedrückt werden (*klicken*). Anschließend erscheint wieder das Fenster, in dem eine Benutzererkennung einzugeben ist, wobei bereits allerdings die zuletzt eingegebene Kennung angegeben ist. Diese kann übernommen werden (`<cr>` tippen) oder gelöscht werden (`<bs>` tippen), wonach erneut eine Benutzererkennung eingegeben werden muss. Anschließend wird wieder nach dem Passwort gefragt.

Nach erfolgreicher Anmeldung erscheint der CDE-Standard-Bildschirm. Er verfügt über 4 virtuelle Bildschirme — Arbeitsbereiche genannt — zwischen denen man hin- und her schalten kann.

Auf dem ersten Arbeitsbereich sind bereits einige Fenster geöffnet, welche zu einem Einführungs- und Hilfeprogramm zu CDE gehören.

Auf jedem Arbeitsbereich befindet sich unten quer eine Hauptanzeige, von der aus

gewisse Standardanwendungen gestartet werden können und von der aus man gewisse Steuerungsfunktionen ausführen kann.

Durch Betätigen des Knopfes mit *Exit* als Aufschrift auf der Hauptanzeige wird die Sitzung beendet (Abmeldung vom System) und kurze Zeit später erscheint wiederum das Fenster, in welchem wieder nach einer Benutzerkennung gefragt wird.

Im Praktikum wird gezeigt, wie ein Benutzer seine persönliche Abreitsoberfläche einstellen kann und wie er dafür sorgen kann, dass beim nächsten Anmelden seine persönliche Umgebung wieder vorliegt.

Insbesondere wird gezeigt, wie ein Terminalfenster geöffnet werden kann. Terminalfenster sind Fenster, in denen gewöhnliche UNIX-Befehle eingeben und dann vom System ausgeführt werden. Es können gleichzeitig mehrere Terminalfenster geöffnet sein, jedoch ist höchstens eins "aktiviert" (erkennbar am farblich anders unterlegtem Rahmen). Man aktiviert ein Terminalfenster, indem man den Mauszeiger ins Fenster positioniert und mit der linken Maustaste klickt.

Im Folgenden wird vorwiegend über derartige Terminalfenster mit dem Rechner gearbeitet.

1.2.2 Die Shell als Kommandointerpreter

Ist ein Terminalfenster geöffnet, so läuft für dieses Fenster ein Kommandointerpreter, der Eingaben aus "diesem Fenster" (d.h. Tastatureingaben bei aktiviertem Fenster) entgegennimmt, diese analysiert und interpretiert und entsprechende Befehle an den Rechner zur Ausführung weitergibt. Die Reaktionen des Rechners auf den abgesetzten Befehl (Ausgaben) erscheinen i. Allg. ebenfalls in diesem Fenster.

Ein derartiger Kommandointerpreter heißt *Shell*. Auf UNIX-Systemem gibt es unterschiedliche solcher Shells, standardmäßig die Bourne-Shell (**sh**), die C-Shell (**chs**) und die Korn-Shell (**ksh**). Unsere Rechner sind so eingestellt, dass i. Allg. eine **ksh** als Kommandointerpreter aktiv ist.

Prompt:

Ist die **ksh** zur Entgegennahme eines Befehls bereit, so meldet sie sich dem sog. *Prompt*, d.h. es wird dann immer eine gewisse Zeichenkette auf dem Terminalfenster ausgegeben.

Diese Prompt-Zeichenkette kann von jedem Benutzer eingestellt werden. Auf unseren Rechnern hat sie standardmäßig zunächst folgende Form:

Betriebssystemname:Rechnername:[Befehlsnummer]\$

Zwischen Rechnername und Befehlsnummer kann weitere Information (Hinweis auf das aktuelle Verzeichnis, siehe Abschnitt 1.2.4) stehen.

Allgemeine Kommandosyntax:

Hat die Shell ihr Prompt ausgegeben, so kann ein neues Kommando eingegeben werden. Dieses wird dann von der Shell ausgeführt und nach Beendigung der Ausführung wird wiederum das Prompt ausgegeben (zwischenzeitlich erscheinen die Ausgaben des ausgeführten Kommandos im Fenster).

Kommandos sind eingegebene Zeichenketten, welche durch ein `<cr>` beendet an die

Shell “abgeschickt“ werden.

Auf UNIX-Systemen wird prinzipiell zwischen Groß- und Kleinschreibung unterschieden, d.h. die Zeichenketten `ls`, `Ls`, `lS` und `LS` werden unterschieden (wobei nur die erste zu einem sinnvollen UNIX-Kommando führt, s.u.!).

Ein UNIX-Kommando kann i. Allg. auf unterschiedliche Art und Weise aufgerufen werden: mit oder ohne Argumente, mit oder ohne Optionen:

Kommandoname {Optionen} {Argumente} <cr>

(Die geschweiften Klammern um Optionen und Argumente gehören nicht zur Eingabe, sondern sollen deutlich machen, dass das, was in ihnen steht, angegeben werden kann oder nicht!)

Argumente beziehen sich auf “Objekte“, auf welche das Kommando wirken soll oder welche zur Ausführung des Kommandos notwendig sind. Optionen beziehen sich auf die Art und Weise, wie das Kommando auf die Objekte einwirken soll.

I. Allg. können hinter dem Kommandonamen zunächst Optionen angegeben werden — dies sind meist Zeichen oder Zeichenketten, denen ein Minuszeichen vorangestellt ist — und anschließend können dann die Argumente aufgeführt werden — das sind meist gewöhnliche, durch Leerzeichen voneinander getrennte Zeichenketten. Das komplette Kommando ist dann durch <cr> abzuschließen und “abzuschicken“.

Sind keine Optionen angegeben, so läuft das Kommando nach einer Standardeinstellung ab.

Sind keine Argumente angegeben, so wirkt das Kommando auf (vom Kommando abhängende) Standardobjekte.

Beispiel:

Das später genauer vorgestellte Kommando `ls` hat (u.a.) die Optionen `l`, `a` und `t` und es können kein, ein oder mehrere Namen als Argumente angegeben werden.

Es sind somit (u.a.) folgende Aufrufe möglich:

<code>ls<cr></code>	Aufruf ohne Optionen und Argumente, es läuft die einfachste Form des Kommandos ab.
<code>ls name<cr></code>	Aufruf mit einem Argument. Das Kommando bezieht sich auf den angegebenen Namen.
<code>ls name1 name2</code>	Aufruf mit zwei Argumenten. Das Kommando bezieht sich auf die angegebenen Namen.
<code>ls -l<cr></code>	Aufruf mit der <code>l</code> -Option ohne Argumente.
<code>ls -a<cr></code>	Aufruf mit der <code>a</code> -Option ohne Argumente.
<code>ls -l -a<cr></code>	Aufruf mit der <code>l</code> - und der <code>a</code> -Option, ohne Argumente.
<code>ls -la<cr></code>	Mehrere Optionen können zusammengefasst werden — dieser Aufruf ist gleichwertig zum vorherigen.
<code>ls -l name<cr></code>	Aufruf mit der <code>l</code> -Option und einem Argument.
<code>ls -tla name1 name2<cr></code>	Aufruf mit drei Optionen und zwei Argumenten.

Zu den gängigen UNIX-Befehlen kann man sich eine Hilfestellung ausgeben lassen, in der die möglichen Optionen und Argumente erläutert werden. Hierzu ist der Befehl:

`man befehlsname<cr>`

einzugeben, wobei *befehlsname* der Name des Befehls ist, zu dem die Hilfestellung erwünscht ist. (Nicht erschrecken — die Ausgabe der Hilfestellung ist in Englisch und einige UNIX-Befehle haben über einige dutzend verschiedene Optionen!)

1.2.3 Passwort ändern

Jeder Benutzer kann für seine Benutzerkennung ein neues Passwort vereinbaren. Dies geschieht durch Eingabe des Kommandos: `passwd<cr>` in einem Terminalfenster.

Es wird dann zunächst vom System nach dem bisherigen (alten) Passwort gefragt, welches einzugeben ist. Anschließend wird zweimal nach dem (gleichen) neuen Passwort gefragt. Sowohl das alte als auch das neue Passwort werden nicht im Terminalfenster angezeigt, so dass man diese wiederum “blind“ eingeben muss.

Passworte sollten aus mindestens 6 Zeichen bestehen und mindestens ein Sonderzeichen (kein Buchstabe) enthalten. Natürlich sollten Passworte nicht zu einfach zu erraten sein. Zu kurze oder allzu einfache neue Passworte werden vom System nicht akzeptiert (Meldungen des Systems beachten!).

Eine der ersten Tätigkeiten im Praktikum wird sein, das vom Systemverwalter vergebene vorläufige Passwort zu ändern.

Aus Sicherheitsgründen empfiehlt es sich, das Passwort in regelmäßigen Zeitabständen zu erneuern.

1.2.4 Das UNIX-Dateisystem

Ein Rechner verarbeitet Daten unterschiedlichster Art. Texte, Tabellen, Bilder, “Sounds“ und Programme sind Daten, d.h. Folgen von einzelnen Bit’s.

Derartige Bitfolgen sind unter einem Namen als *Dateien* (*Files*) auf Festplatten eines Rechners abgespeichert.

Je nach Inhalt (genauer: Art und Weise, wie die Bitfolgen in der Datei interpretiert werden) unterscheidet man Textdateien (Inhalt auf dem Bildschirm ansehbar), Programmdateien (Inhalt als Befehlsfolge auf dem Rechner ablaufbar) oder allgemeine Binärdateien (etwa Bilder oder “Sounds“ — auf solche Dateien kann man sinnvoll nur mit speziellen Programmen, etwa Bildbetrachter oder “Sound-Tools“ zugreifen). Eifrige Benutzer besitzen einige hundert Dateien.

Zum Betriebssystem selber gehören einige tausend Dateien.

Zur Strukturierung und zur Erleichterung des Umgangs mit diesen mehreren tausend Dateien (und entsprechenden Namen) werden die Dateien in unterschiedliche “Bereiche“ (synonym: *Dateiverzeichnisse*, *Verzeichnisse*, *Directories*, *Ordner*) eingeteilt und diese Bereiche werden hierarchisch, baumartig organisiert:

Ein Verzeichnis kann mehrere Dateien (mit verschiedenen Namen) sowie mehrere Unterverzeichnisse enthalten. In diesen Unterverzeichnissen können wiederum andere Dateien und weitere Unterverzeichnisse enthalten sein.

Jedes Verzeichnis (mit Ausnahme des sogenannten *Wurzelverzeichnisses*) ist in genau einem anderen Verzeichnis (sein sog. *Vaterverzeichnis*) als Unterverzeichnis enthalten.

Neben den eigentlichen Dateien (mit Bitfolgen als Inhalt) existieren somit zur Strukturierung in einem UNIX-Dateisystem auch noch Dateiverzeichnisse und diese Dateiverzeichnisse haben natürlich auch Namen und sind irgendwie auch auf den Festplatten des Rechners gespeichert.

Trägt man die “Enthalten in“-Beziehung der Dateien und Dateiverzeichnisse als Graph auf, so erhält man einen “Baum“ (d.h. es kommen keine Zyklen (Kreise) vor) mit einer Wurzel, dem Wurzelverzeichnis /:

Begriffe:

- Wurzelverzeichnis:

Wurzel des Dateibaums. Einziges Verzeichnis ohne Väterverzeichnis.

- Aktuelles Verzeichnis:

Verzeichnis, in dem sich ein Benutzer (genauer: dessen Shell) “im Augenblick befindet“. Einfache dateiorientierte Kommandos beziehen sich auf Dateien und Unterverzeichnisse in diesem aktuellen Verzeichnis. Die hier liegenden Dateien sind direkt unter ihrem einfachen Namen zugänglich.

- Heimatverzeichnis oder Home-Verzeichnis oder Login-Verzeichnis:

Jeder Benutzer hat sein persönliches Verzeichnis, in dem er Dateien und Unterverzeichnisse erzeugen darf. Nach dem Anmelden ist dieses Heimatverzeichnis das aktuelle Verzeichnis, d.h. der Benutzer befindet sich nach dem Anmelden (bzw. Öffnen eines Terminalfensters) im Heimatverzeichnis.

Bei vernetzten Rechnern braucht das Heimatverzeichnis (oder auch ein anderes Verzeichnis) physikalisch gar nicht lokal auf dem Rechner zu liegen, an dem gearbeitet wird. Mittels *NFS* (*N*etwork *F*ile *S*ystem) können Teile des (physikalischen) Dateibaums eines Rechners (logisch) in den Dateibaum eines anderen Rechners eingehängt werden.

Dies wird u.a. bei unseren Heimatverzeichnissen angewendet.

- absoluter Pfad:

Jede Datei (bzw. jedes Dateiverzeichnis) ist an eindeutiger Stelle in den Dateibaum eingehängt. Schreibt man vom Wurzelverzeichnis ausgehend, jeweils durch ein / getrennt hintereinander die Namen aller Zwischenverzeichnisse auf dem Weg bis zur anvisierten Datei (bzw. zum anvisierten Dateiverzeichnis) und schließlich den Dateinamen (bzw. Verzeichnisnamen) auf, so erhält man den absoluten Pfad zu dieser Datei (bzw. Dateiverzeichnis).

Der absolute Pfad zu meinem Heimatverzeichnis `ax006ha` im obigen Schaubild lautet somit `/home/ax006ha` und der absolute Pfad zu der Datei `prog1` im Unterverzeichnis `bin` meines Heimatverzeichnisses `ax006ha` lautet somit: `/home/ax006ha/bin/prog1`

Jede Datei und jedes Unterverzeichnis ist durch den zugehörigen absoluten Pfad eindeutig identifiziert. (Es kann unterschiedliche Dateien mit dem gleichen Namen geben — es kann aber nicht unterschiedliche Dateien mit dem gleichen absoluten Pfad geben!)

Absolute Pfadangaben fangen immer mit einem / an!

- relativer Pfad:

Auch relative Pfade dienen zur Identifizierung von Dateien oder Verzeichnissen.

Im Gegensatz zu absoluten Pfaden ist jedoch nicht das Wurzelverzeichnis Ausgangspunkt für die Pfadangabe, sondern das aktuelle Verzeichnis. Dieses wird in der relativen Pfadangabe nicht oder in der Form `./` aufgeführt.

Ist etwa das Verzeichnis `/home/ax006ha` das (augenblicklich) aktuelle, so ist `bin/prog1` oder `./bin/prog1` der relative Pfad zur Datei `/home/ax006ha/bin/prog1` (absoluter Pfad).

Ist jedoch `/home` das augenblicklich aktuelle Verzeichnis, so lautet der relative Pfad zur gleichen Datei: `ax006ha/bin/prog1` oder `./ax006ha/bin/prog1`.

Relativen Pfadangaben können auch mit `../` oder `../../` usw. beginnen. Durch `..` wird das Väterverzeichnis angesprochen. Somit bedeutet `../` Väterverzeichnis des aktuellen und `../../` Väterverzeichnis des Väterverzeichnisses des aktuellen usw..

Ist etwa `/home/ax006ha/csrce` das aktuelle Verzeichnis und soll die gleiche Datei wie in den obigen Beispielen angesprochen werden, so ist dies mit dem relativen Pfad `../bin/prog1` möglich.

Relative Pfadangaben beinhalten mindestens ein `/`-Zeichen, fangen aber nicht mit diesem an.

Namen von Dateien und Dateiverzeichnissen können weitgehend beliebig sein, insbes. dürfen sie länger als 8 Zeichen sein und können mehrere, etwa durch Punkte getrennte Bestandteile haben. Groß- und Kleinschreibung werden unterschieden.

Leider ist am Namen nicht automatisch erkennbar, ob es sich um eine gewöhnliche Datei oder um ein Dateiverzeichnis handelt bzw. welchen Inhalt eine gewöhnliche Datei hat.

An dieser Stelle sind wir in der Lage, die Zusatzinformation im `ksh`-Standardprompt zwischen Rechnername und Befehlsnummer zu erläutern:

- Befindet die Shell sich in einem Verzeichnis unterhalb des Heimatverzeichnisses, so wird im Prompt zwischen Rechnername und Befehlsnummer der relative Pfad ausgehend vom Heimatverzeichnis zum aktuellen Verzeichnis angegeben.
- Befindet man sich nicht unterhalb des Heimatverzeichnisses, so wird dort der absolute Pfad zum aktuellen Verzeichnis angegeben.

1.2.5 Einfache Dateiorientierte Kommandos

Pfad zum aktuellen Verzeichnis ausgeben lassen:

Kommando: `pwd<cr>`
(*print working directory*)

Aktuelles Verzeichnis wechseln:

Kommando: `cd neues_Verzeichnis<cr>`
(*change directory*), wobei `neues_Verzeichnis` entweder ein direktes Unterverzeichnis des aktuellen Verzeichnisses oder ein absoluter oder relativer Pfad zu einem anderen Verzeichnis ist. Dieses direkte Unterverzeichnis bzw. das durch den Pfad angegebene andere Verzeichnis wird zum neuen aktuellen Verzeichnis.

Weitere Formen:

`cd ..<cr>` Wechsel ins Väterverzeichnis des aktuellen Verzeichnisses.
(Leerzeichen zwischen `cd` und `..` beachten!)

`cd<cr>` Heimatverzeichnis wird aktuelles Verzeichnis.

Inhaltsverzeichnis eines Verzeichnisses ausgeben lassen:

Kommando: `ls<cr>`

(*list*). Dieses Kommando (ohne Argumente) gibt eine alphabetisch sortierte Liste der Namen der Unterverzeichnisse und Dateien im aktuellen Verzeichnis aus.

Ein als Argument angegebener Name eines direkten Unterverzeichnisses oder die Pfadangabe (relativ oder absolut) eines Verzeichnisses bewirkt, dass eine Liste der Dateien und Unterverzeichnisse des angegebenen Verzeichnisses ausgegeben wird.

Ein als Argument angegebener Name einer Datei im aktuellen Verzeichnis oder die Pfadangabe zu einer Datei bewirkt, dass der Name dieser Datei ausgegeben wird.

Es können mehrere Argumente angegeben werden.

Das `ls`-Kommando verfügt über eine ganze Reihe von Optionen, u.a.:

- a Es werden auch die Namen sog. *versteckter* Dateien bzw. Dateiverzeichnisse in der Liste ausgegeben. Diese werden normalerweise nicht mit ausgegeben. (Die Namen versteckter Dateien bzw. Dateiverzeichnisse beginnen mit einem Punkt!)
- t Die Liste der Namen ist normalerweise alphabetisch sortiert. Die Option `-t` sorgt dafür, dass die Liste nach dem Modifikationsdatum der Dateien bzw. Verzeichnisse sortiert ist: der Name der zuletzt veränderten Datei bzw. des zuletzt veränderten Verzeichnisses wird zuerst ausgegeben.
- l Es wird neben den Namen weitere Information zu den Dateien bzw. Unterverzeichnissen ausgegeben.

Diese letzte Option ist so wichtig, dass wir uns ein wenig näher mit diesem Kommando `ls -l<cr>` beschäftigen wollen.

Jeder Benutzer eines UNIX-Rechners hat einen Benutzernamen (gleich Namen, mit dem er sich beim Einloggen anmeldet) und eine Benutzernummer (UID).

Jeder Benutzer ist einem *Team* von Benutzern zugeteilt. Dies ist eine Menge von Benutzern, welche an ein und demselben Projekt arbeiten. Ein solches *Team* heißt *Gruppe* und jede Gruppe hat einen Gruppennamen und eine Gruppennummer (GID).

Eigene Benutzernamen, Benutzernummer, Gruppennamen und Gruppennummer kann man sich mit dem Kommando: `id<cr>` auf dem Bildschirm ausgeben lassen.

Jede Datei und jedes Dateiverzeichnis hat einen Eigentümer — das ist derjenige Benutzer, der die Datei bzw. das Verzeichnis erzeugt hat.

Bedeutung der Rechte:

- Bei gewöhnlichen Dateien:
 - Leserecht: Erlaubnis, sich den Inhalt der Datei anzusehen (etwa mit dem Kommando `cat`, siehe Seite 15) oder eine Kopie der Datei zu erstellen (`cp`-Kommando, siehe Seite 13).
 - Schreibrecht: Erlaubnis, den Inhalt der Datei zu ändern (etwa durch Überkopieren mit dem Kommando: `cp`).
 - Ausführungsrecht: Erlaubnis, den Namen der Datei als Kommando einzugeben. (Der Inhalt der Datei sollte hierbei ein übersetztes Programm oder eine Kommando-Prozedur sein!)
- Bei Dateiverzeichnissen:
 - Leserecht: Erlaubnis, sich den Inhalt des Dateiverzeichnisses (etwa mit dem `ls`-Kommando, siehe Seite 11) anzusehen.
 - Schreibrecht: Erlaubnis, in diesem Dateiverzeichnis neue Dateien oder Unterverzeichnisse anzulegen **und** Dateien und (leere) Unterverzeichnisse in diesem Verzeichnis zu löschen.
 - Ausführungsrecht: Erlaubnis, dieses Dateiverzeichnis zum aktuellen Verzeichnis zu machen (d.h. in dieses Verzeichnis hineinzuschalten).

Neue Verzeichnisse erzeugen:

Kommando: `mkdir name<cr>`

(*make directory*). Im einfachsten Fall ist **name** ein neuer Name (kein Pfad, d.h. es kommt kein `/` vor). Im aktuellen Verzeichnis wird dann ein neues Verzeichnis angelegt und dieses Verzeichnis erhält den angegebenen Namen.

Das Argument **name** kann aber auch der relative oder absolute Pfad zu dem neu zu erzeugenden Verzeichnis sein. Hierbei muss allerdings das Vaterverzeichnis des zu erzeugenden Verzeichnisses bereits vorhanden sein.

Auf alle Fälle muss der Aufrufer das Schreibrecht für das Vaterverzeichnis des neu zu erstellenden Verzeichnisses besitzen.

Schließlich kann **name** auch eine durch Leerzeichen getrennte Liste von neu anzulegenden Verzeichnissen (Namen oder Pfade) sein — es werden dann gleich mehrere neue Verzeichnisse angelegt.

Leere Verzeichnisse löschen:

Kommando: `rmdir name<cr>`

(*remove directory*). Ist das angegebene Verzeichnis (direktes Unterverzeichnis des aktuellen oder Pfad zu einem Verzeichnis) leer und besitzt der Aufrufer das Schreibrecht für das Vaterverzeichnis des angegebenen Verzeichnisses, so wird dieses gelöscht und aus dem Dateibaum entfernt.

Auch hier kann **name** eine Liste von Verzeichnissen sein — diese alle werden, falls sie leer sind und entsprechende Rechte vorliegen, gelöscht.

Dateien löschen:

Kommando: `rm name<cr>`

(*remove*). Hierbei ist **name** der Name einer Datei im aktuellen Verzeichnis oder eine Pfadangabe zu einer Datei (oder eine Liste von Namen oder Pfadangaben). Besitzt der Aufrufer das Schreibrecht für das Verzeichnis, in dem die angegebene Datei liegt, so wird diese gelöscht und aus dem Dateibaum entfernt. (Beachte: das Schreibrecht für die zu löschende Datei ist hierzu **nicht** erforderlich!!!)

Dateien kopieren

Kommando: `cp quelle ziel<cr>`

(*copy*). Der `cp`-Befehl dient zum Kopieren von Dateien.

In der einfachsten Form ist **quelle** der Name (bzw. der Pfad zu) einer existierenden Datei und **ziel** der Name (bzw. der Pfad zu) einer noch nicht existierenden Datei. Diese neue Datei wird erzeugt und bekommt als Inhalt eine Kopie der existierenden Datei.

Ist die Datei **ziel** bereits vorhanden, so wird deren Inhalt überschrieben. (Hierbei müssen **quelle** und **ziel** verschiedene Dateien sein — eine Kopie auf sich selbst ist nicht möglich!)

Ist das Ziel ein bereits existierendes Verzeichnis (Name eines direkten Unterverzeichnisses des aktuellen Verzeichnisses oder Pfad zu einem Verzeichnis oder `.` für das aktuelle Verzeichnis oder `..` für dessen Vaterverzeichnis), so wird die Kopie von **quelle** unter dem gleichen Namen wie der Dateiname der Quelle im angegebenen Zielverzeichnis angelegt.

Ist das Ziel ein Verzeichnis, so kann **quelle** selber auch eine Liste von Dateinamen sein — es werden im angegebenen Zielverzeichnis Kopien der in der Liste angegebenen Dateien mit ihren Originalnamen erzeugt.

Zum Kopieren einer Datei ist das Leserecht für die zu kopierende Datei und das Schreibrecht für das Verzeichnis erforderlich, in dem die Kopie zu stehen kommen soll. Die Kopie hat den Aufrufer des Kopierbefehls als Eigentümer.

Dateien bzw. Dateiverzeichnisse umbenennen

`mv quelle ziel<cr>`

(*move*). Der Aufruf dieses Befehls ist analog zu dem von `cp`. Im Gegensatz zu diesem ist jedoch das Original nach Ausführung des `mv`-Befehls nicht mehr vorhanden, sondern nur noch die "Kopie".

(Natürlich wird keine echte Kopie der Datei erstellt und anschließend das Original gelöscht, sondern das Original wird aus dem Dateibaum gelöst und an anderer Stelle — ggf. mit anderem Namen — wieder eingehängt! Daher der Name *move*, *bewegen*!) Zur Ausführung dieses Befehls ist das Schreibrecht für das Quellverzeichnis (wo das Original vor dem Befehl stand) und Zielverzeichnis (wo es anschließend stehen soll) erforderlich.

In der einfachsten Form, wo **quelle** der Name einer Datei oder eines direkten Unterverzeichnisses des aktuellen Verzeichnisses (also keine Pfadangabe) ist und **ziel** ein neuer Name (ebenfalls kein Pfad), bewirkt der `mv`-Befehl ein Umbenennen dieses Objektes im aktuellen Verzeichnis.

Wie bei `cp` kann auch bei `mv`, wenn das Ziel ein bereits existierendes Verzeichnis ist, **quelle** eine Liste von zu “bewegenden“ Objekten sein. Alle in **quelle** angegebenen Objekte werden unter ihrem Originalnamen in das angegebene Zielverzeichnis umgehängt.

Eine mittels `mv` im Dateibaum umgehängte Datei (bzw. Verzeichnis) behält ihren ursprünglichen Eigentümer, d.h. der Aufrufer von `mv` wird nicht zum neuen Eigentümer!

Dateiinhalte auf dem Bildschirm ansehen

Bevor Sie sich Dateiinhalte auf dem Bildschirm ansehen, sollten Sie folgendes beachten:

- Um sich den Inhalt einer Datei ansehen zu können, ist formal das Leserecht für diese Datei erforderlich. Ist das Leserecht nicht gewährt, so verbietet das System einen entsprechenden Versuch. Das Ansehendürfen ist jedoch nicht der einzige Zweck des Leserechtes — es ist etwa auch erforderlich, um etwa die Datei kopieren zu können.
- Ist das Leserecht gewährt, so ist zwar vom System her prinzipiell erlaubt, sich den Dateinhalt anzusehen — doch unter gewissen Umständen verbietet dieses der gesunde Menschenverstand. Denn nicht bei jedem Dateinhalt ist dieses sinnvoll, etwa bei übersetzten Programmen oder “Sound“-Dateien oder Ähnlichem. Bei solchen Dateien kann das Leserecht zwar gewährt sein, um etwa das Kopieren der Datei zu ermöglichen, trotzdem sollte man nicht versuchen, sich den Dateinhalt direkt anzusehen!

Da unter UNIX Dateinamen weitgehend beliebig sind, kann man u. U. vom Dateinamen nicht auf den Dateinhalt schließen. (Es gibt zwar gewisse Konventionen — etwa Endung `.txt` für Textdateien, `.c` für C-Quellprogramme, `.f` oder `.for` für FORTRAN-Quellprogramme, `.o` für übersetzte Objektprogramme — doch kann man sich nicht darauf verlassen, dass jeder Benutzer sich an diese Konventionen hält.)

Bevor man sich den Inhalt einer fremden Datei ansieht, für die man das Leserecht hat, sollte man versuchen herauszubekommen, ob der Dateinhalt für die Bildschirmausgabe geeignet ist.

Dies kann mit dem Kommando: `file Dateiname<cr>` geschehen.

Dieses Kommando bewirkt, dass das System den Dateinhalt mal durchgeht und anhand verschiedener Merkmale untersucht, um was für einen Inhalt es sich handeln könnte. Es wird dann eine entsprechende Meldung ausgegeben.

Die hierbei möglichen Meldungen: `ASCII-text`, `englisch-text`, `script`, `c-program-text` deuten auf “ausgebbar“ hin, d.h. die entsprechenden Dateiinhalte kann man sich gefahrlos auf dem Bildschirm ausgeben lassen.

Die Meldungen `data`, `ELF...` hingegen weisen darauf hin, dass der Dateinhalt nicht zur Bildschirmausgabe geeignet ist.

Das `file`-Kommando kann auch auf Verzeichnisse angewendet werden — es erscheint die Meldung: `directory`

Im Folgenden wird von Dateien ausgegangen, für die man das Leserecht besitzt und welche für die Bildschirmausgabe geeignet sind.

Das Kommando: `cat datei<cr>`

gibt den Inhalt der angegebenen Datei (oder Dateien, es können mehrere angegeben werden) auf dem Bildschirm aus!

Sind mehrere Dateien angegeben, so werden deren Inhalte nacheinander ausgegeben. Bei mehreren Dateien oder bei längeren Dateien rauscht der Inhalt über den Bildschirm, so dass man kaum etwas lesen kann!

Das Kommando `more datei<cr>`

gibt den Inhalt der angegebenen Datei(en) seitenweise auf dem Bildschirm aus, d.h. zunächst wird nur die erste Seite der (ersten) Datei ausgegeben. Durch gewisse Tastendrucke kann der weitere Ablauf gesteuert werden:

Taste	Wirkung
<i>Leertaste</i>	nächste Seite wird ausgegeben
<code><cr></code>	nächste Zeile wird ausgegeben
b	vorherige Seite wird ausgegeben (zurückblättern)
q	Ausgabe der Datei wird beendet

Das Kommando: `tail datei<cr>`

gibt standardmäßig die letzten 10 Zeilen der angegebenen Datei auf dem Bildschirm aus. Die Anzahl der auszugebenden Zeilen kann mit einer Option gesteuert werden:

`tail -20 datei<cr>` gibt etwa die letzten 20 Zeilen aus.

Das Kommando: `head datei<cr>`

funktioniert analog zum `tail`-Kommando, es gibt jedoch die ersten 10 (oder die als Option angegebene Anzahl) Zeilen der Datei auf dem Bildschirm aus.

1.2.6 Ein- und Ausgabeumlenkung

Jedes UNIX-Kommando verfügt über drei Kanäle, mit denen es mit dem Benutzer kommuniziert.

Dies sind

Name	Nr.	Bedeutung
<i>stdin</i>	0	Standardeingabe; von ihr nimmt ein ablaufendes Kommando ggf. weitere Eingaben entgegen. Die Standardeingabe ist im Normalfall die Tastatur.
<i>stdout</i>	1	Standardausgabe; hier gibt das ablaufende Kommando seine Meldungen aus. Die Standardausgabe ist im Normalfall der Bildschirm.
<i>stderr</i>	2	Standardfehlerausgabe; hier gibt das Kommando gewöhnlich Fehlermeldungen aus. Die Standardfehlerausgabe ist im Normalfall ebenfalls der Bildschirm.

Beim Aufruf eines Kommandos kann man diese Kanäle umlenken (auch gleichzeitig).

Umlenkung der Standardeingabe:

Gibt man vor dem "abschließenden" `<cr>` eines Kommandos `< dateiname` ein, so

liest dieses Kommando weitere Eingaben nicht von der Tastatur, sondern von der angegebenen Datei. In dieser Datei muss nun die für das Kommando bestimmte Eingabe so stehen, wie sie auch “interaktiv“ von der Tastatur einzugeben wäre. (Jedes `<cr>` in der interaktiven Eingabe entspricht einem Zeilenende in der Datei.)

Steht in der Datei mehr, als das Kommando zur Abarbeitung liest, so ist dies unerheblich. Hat das Kommando alle Zeilen der Datei gelesen und verlangt nach weiterer Eingabe, so erhält es Kenntnis darüber, dass das Ende der Eingabe (*EOF*) erreicht wurde. Hierauf reagieren unterschiedliche Kommandos unterschiedlich.

Auch in der interaktiven Verwendung eines Kommandos (ohne Eingabeumlenkung) kann (auf UNIX-Systemen) von der Tastatur aus das Ende der Eingabe durch “`<ctrl>d`“ “eingegeben“ werden. (Auf MSDOS-Rechnern geschieht dies i. Allg. durch “`<ctrl>z`“.)

Umlenkung der Ausgabe:

Zur Umlenkung der Ausgabe stehen drei Möglichkeiten zur Verfügung:

Kommando	Wirkung
<code>kommando > datei<cr></code>	Die Standardausgabe erscheint nicht auf dem Bildschirm, sondern wird in die angegebene (neue) Datei geschrieben. Eine existierende Datei wird i. Allg. nicht überschrieben. Man muss natürlich über das Recht zur Erstellung der Datei verfügen.
<code>kommando > datei<cr></code>	Wie oben, jedoch darf die Datei bereits existieren. Diese wird durch die Ausgabe des Kommandos überschrieben. (Schreibrecht für die Datei erforderlich!)
Kommando	Wirkung
<code>kommando >> datei<cr></code>	Die Standardausgabe des Kommandos wird an den Inhalt der angegebenen Datei angehängt. (Schreibrecht für die Datei erforderlich!)

Den Umlenkungszeichen “`>`“, “`>|`“ und “`>>`“ kann die Nummer des Ausgabekanals vorangestellt werden, etwa “`1>`“, “`1>|`“ und “`1>>`“ für die Standardausgabe (*stdout*), aber auch “`2>`“, “`2>|`“ und “`2>>`“ für die Standardfehlerausgabe (*stderr*). Ohne Angabe der Kanalnummer bezieht sich die Ausgabeumlenkung immer auf die Standardausgabe.

Interessiert die Ausgabe eines Kommandos (oder Programms) nicht und will man sie auch nicht am Bildschirm über sich ergehen lassen, so kann man die Ausgabe nach `/dev/null` umlenken.

Man kann auch die Standardfehlerausgabe und die Standardausgabe gleichzeitig (auf verschiedene Dateien) umlenken, etwa:

```
kommando 1> datei1 2> datei2<cr>
```

Die gleichzeitige Umlenkung auf eine Datei ist ebenfalls möglich:

```
kommando 1> datei1 2>&1<cr>
```

Hier wird zunächst (`1> datei1`) die Standardausgabe auf die angegebene Datei umge-

lenkt und anschließend die Standardfehlerausgabe auf die Standardausgabe umgelenkt (2>&1).

Anwendung der Ausgabeumlenkung beim `cat`-Kommando:

Mit Ausgabeumlenkung kann das `cat`-Kommando zum Kopieren von Dateien missbraucht werden:

```
cat datei > ziel<cr>
```

Hierbei ist `datei` die zu kopierende Datei und `ziel` ein neuer Name.

Es können auch mehrere Dateien aneinandergehängt werden:

```
cat datei1 datei2 datei3 >> ziel<cr>
```

Nach diesem Befehl steht in der (neuen) Datei `ziel` hintereinander der Inhalt der drei (auch weiterhin vorhandenen) Dateien `datei1`, `datei2` und `datei3`.

Mit dem `cat`-Kommando kann zusammen mit Ausgabeumlenkung auch neue Dateien mit neuem Inhalt erzeugen. Zunächst kann das `cat`-Kommando auch ohne einen Dateinamen als Argument aufgerufen werden. Die Voreinstellung von `cat` ist, dass in diesem Fall der "Inhalt" der Standardeingabe (also Tastatur) auf dem Bildschirm ausgegeben wird. Da alles, was auf der Tastatur eingegeben wird, vom System (nicht vom `cat`-Kommando) sowieso auf dem Bildschirm ausgegeben wird (mit Ausnahme der Passworeingabe), erscheint alles eingetippte jetzt zweimal auf dem Bildschirm, einmal vom System her und einmal durch das `cat`-Kommando. Diese zweimalige Ausgabe aller eingetippten Zeichen wird durch die Eingabe von `<ctrl> d` ("Dateiende") beendet.

Durch Umlenkung der Ausgabe können nun einfache Textdateien erstellt werden:

```
cat > neuedatei<cr>
```

Alles, was nun (bis zum nächsten `<ctrl> d`) eingetippt wird, erscheint vom System her auf dem Bildschirm und wird durch das `cat`-Kommando mit Umlenkung auch in die angegebene neue Datei geschrieben.

1.2.7 Metazeichen, Cut & Paste

Bei dateiorientierten Kommandos können Namen von (bereits vorhandenen) Dateien oder Verzeichnissen (u.a.) durch die Metazeichen `*` und `?` abgekürzt werden, d.h. der Name braucht nicht komplett angegeben zu werden. Ein `*` in der Angabe eines Namens wird von der Shell als "beliebige Zeichenkette" und ein `?` als "ein beliebiges Zeichen" interpretiert. (Beachte, die Shell interpretiert die Metazeichen in einem Befehl, nimmt Ersetzungen vor und gibt den interpretierten Befehl ans Betriebssystem weiter!)

Soll etwa eine Datei `gcc-2.7.2.tar.gz` im aktuellen Verzeichnis gelöscht werden, so kann hierzu der Befehl:

```
rm gcc-2.7.2*<cr>
```

verwendet werden. Doch Vorsicht, sollten im aktuellen Verzeichnis mehrere Dateien existieren, deren Namen mit `gcc-2.7.2` beginnen, so werden durch obigen Befehl all diese gelöscht! (D.h. `gcc-2.7.2*` wird von der Shell durch die Liste der Dateinamen

im aktuellen Verzeichnis ersetzt, die mit `gcc-2.7.2` beginnen!)

Der `*` kann mehrmals im Namen auftreten, etwa: `h*.*` bedeutet: alle Dateien, deren Namen mit einem `h` beginnen und mindestens einen Punkt enthalten.

Ein `*` am Anfang eines Namens bezieht sich nicht auf Dateien oder Verzeichnisse, deren Namen mit einem Punkt beginnen (versteckte Dateien bzw. Verzeichnisse).

Das Fragezeichen ist in analoger Weise zu verwenden. Existieren beispielsweise im aktuellen Verzeichnis die Dateien `kap1.tex`, `kap2.tex` und `kap3.tex`, so kann man diese durch den Befehl:

```
rm kap?.tex<cr>
```

auf einmal löschen (wobei auch eine evtl. vorhandene Datei `kapA.tex`, nicht jedoch eine Datei `kap10.tex` gelöscht würde!).

Bei langen Dateinamen kann man sich auch des von der CDE-Oberfläche gebotenen Cut & Paste bedienen.

Positioniert man den Mauszeiger auf den Anfang einer beliebigen im Terminalfenster angezeigten Zeichenkette, drückt und hält die linke Maustaste, zieht den Mauszeiger bis zu Ende der Zeichenkette und lässt dort erst die Maustaste wieder los, so wird diese Zeichenkette schwarz unterlegt und intern abgespeichert (*Cut, Ausschneiden*).

Durch Klicken der mittleren Maustaste wird die gespeicherte Zeichenkette an die Cursorposition des Fensters, in dem sich der Mauszeiger gerade befindet, kopiert (*Paste*).

Will man etwa einen Befehl mit einem langen Dateinamen einer Datei im aktuellen Verzeichnis als Argument ausführen, kann man wie folgt verfahren:

- `ls`-Kommando ausführen
- gewünschten Dateinamen wie oben beschrieben “ausschneiden“ (*Cut*)
- Kommandonamen und Leerzeichen eingeben (Cursor steht hinter dem Leerzeichen)
- mittlere Maustaste klicken (*Paste*) (Dateiname wird vom System an das Kommando angehängt)
- `<cr>` eingeben (erübrigt sich, falls man das Zeilenende mit ausgeschnitten hat!)

1.2.8 Drucken von Dateien

Dateien mit lesbarem Inhalt können auf dem Drucker im Praktikumsraum ausgedruckt werden. Hierzu können die folgenden Befehle verwendet werden:

```
lp -d ljc3 dateiname<cr>    oder
lpr -P ljc3 dateiname<cr>
```

In beiden Fällen wird die angegebene Datei so wie sie ist (ohne weitere Formatierung oder Seitennummerierung) ausgegeben.

Eine etwas komfortablere Ausgabe ermöglicht das Kommando:

```
a2ps -P ljc3 dateiname<cr>
```

Hierbei werden automatisch die Seiten durchnummeriert und die Datei in ein anschauliches Druckbild übertragen und weitergegeben. Standardmäßig erscheinen zwei Seiten der Datei nebeneinander auf einer Seite eines Papierblattes und es wird zweiseitig gedruckt (Vor- und Rückseite).

Durch Optionen kann die Ausgabe beeinflusst werden:

Option	Wirkung
<code>-1</code>	Es wird doppelseitig gedruckt, aber auf jeder Seite des Papiers steht nur eine Seite der Datei.
<code>-s1</code>	Es wird einseitig gedruckt, aber auf der Druckseite sind nebeneinander zwei Seiten der Datei.
<code>-DTumble=true</code>	Bei doppelseitigem Drucken wird die Orientierung der Rückseite um 180° gedreht (je nach Art der Abheftung des Dokumentes erforderlich).

Aus Gründen der Papier-, Kosten- und Zeiteinsparung werden die Praktikumssteilnehmer gebeten, nur wirklich wichtige (und nur kurze) Dokumente auszudrucken.

1.3 Nutzung des Internets

1.3.1 Informationen aus dem Internet

Viele Institutionen, Firmen und auch Privatleute stellen im Internet Informationen öffentlich zur Verfügung.

Ein gängiges *Protokoll* (Regelwerk, Schema) zur Darstellung solcher Informationen im Internet ist **http** (*hypertext-transfer-protocol*). In den Textdokumenten mit den Informationen können Verweise (*Hyper-Links*) auf andere Dokumente eines anderen Rechners aufgenommen werden und durch “Klicken“ mit der Maus auf diesen Verweis erhält man dieses andere Dokument. Auf diese Weise kann man sich von einer Information zur nächsten bewegen (“Surfen im Internet“). Die Menge aller weltweit so verknüpfter Informationen heißt *WWW* (*World-Wide-Web*).

Vorsicht: Einige Psychologen sind der Ansicht, das Surfen im Internet süchtig machen kann! Deshalb sei an dieser Stelle vor einem allzu sorglosen und leichtfertigen Umgang mit dem Internet gewarnt!

Auf unseren Rechnern steht der *Netscape-Navigator* als *WWW-Browser* zur Verfügung, um Informationen aus dem Internet auf dem Bildschirm auszugeben.

Der Aufruf geschieht am besten wie folgt aus einem Terminalfenster heraus:

```
netscape &<cr>
```

Nach einiger Zeit erscheint der Navigator mit einem neuen Fenster und im Terminalfenster kann anderweitig weitergearbeitet werden.

Beim ersten Aufruf des Navigators erscheinen zunächst einige Meldungen und Rückfragen und es sollten einige Voreinstellungen vorgenommen und abgespeichert werden (Näheres im Praktikum!).

Bei den Voreinstellungen kann als erste WWW-Seite die *Homepage* der RWTH-Aachen eingestellt werden, diese wird dann bei jedem weiteren Aufruf von *Netscape* zunächst angezeigt.

Die Homepage eines Informationsanbieters ist die “Einstiegsseite“ zur Information des Anbieters, hier wird i. Allg. eine Übersicht über die vom Anbieter zur Verfügung gestellte Information gegeben und durch Hyperlinks kann dann auf diese Informationen zugegriffen werden.

Die Hyperlinks sind farbig unterlegt und wenn der Mauszeiger auf einen Hyperlink positioniert wird, verändert sich die Form des Mauszeigers. Nach Klicken mit der Maustaste folgt der Browser dem Hyperlink, d.h. er lädt das Dokument, auf welches der Link verweist. Hierbei kann es sein, daß der Rechner, der diese neue Information zur Verfügung stellt (*WWW-Server*), irgendwo anders auf der Welt steht.

Das Laden von solchen Dokumenten kann je nach Auslastung des Netzes und Standort des *WWW-Servers* länger dauern. Solange das Laden eines Dokumentes noch nicht abgeschlossen ist (das Dokument wird hierbei soweit wie möglich bereits teilweise dargestellt!) verändert das *Stop-Icon* im oberen Bereich des Browsers seine Farbe auf rot und im Netscape-Logo erscheinen “fliegende Sternschnuppen“. Durch Klicken auf das *Stop-Icon* wird der Ladevorgang abgebrochen!

Man kann auch direkt (ohne Hyperlink) einen *WWW-Server* “anwählen“. Hierzu

muß man zunächst auf das Wort **file** in der Menüleiste klicken, worauf ein Auswahlmenü “aufklappt“. Hier ist der Menüpunkt **open location** anzuklicken und in dem dann erscheinenden Eingabefenster muß zunächst folgende Zeichenkette **http://** und unmittelbar dahinter der Name des gewünschten *WWW-Servers* eingegeben werden — also etwa

http://www-ma.rz.rwth-aachen.de

für den *WWW-Server* der MA-Ausbildungs-Abteilung des Rechenzentrums. Nach Eingabe von <cr> oder Klicken auf die entsprechende Bestätigung wird dann die Verbindung zur *Homepage* des gewählten *WWW-Servers* hergestellt.

Viele deutsche Firmen, Verlage und andere Institutionen haben einen *WWW-Server* mit dem Namen: **www.name.de**, wobei *name* der Name der Firma bzw. des Verlags bzw. der Institution ist.

In dem zum Wort **file** der Menüleiste gehörenden Auswahlmenü wird auch der Menüpunkt **exit** zur Beendigung des Netscape-Navigators bereitgestellt.

Der Netscape-Browser stellt eine Fülle weiterer Funktionalitäten zur Verfügung, auf die hier nicht weiter eingegangen wird, die aber im Praktikum ausprobiert werden können. (Ggf. Praktikum-Betreuer fragen!)

1.3.2 Elektronische Post, E-Mail

E-Mail ist ein Internet-Dienst zur Verschickung von Nachrichten von einem Benutzer eines (ans Internet angeschlossenen) Rechners zu einem anderen. Die Nachrichten werden hierbei von gewissen Rechnern im Internet zwischengespeichert (sog. *Mail-Server*), so dass der Empfänger im Augenblick gar nicht an seinem Rechner arbeiten muss — der Rechner des Empfängers braucht nicht einmal eingeschaltet zu sein.

In unserem Netz übernimmt der Mail-Server **postc1** die Zwischenspeicherung der an unsere Benutzer eingetroffene E-Mail. Jeder Praktikumssteilnehmer hat eine sog. *E-Mail-Adresse*, unter der er weltweit erreichbar ist. E-Mail-Adressen haben allgemein die Form:

benutzerkennung@internetname_des_rechners

und somit bei uns:

benutzerkennung@postc1.rz.rwth-aachen.de

Jeder Praktikumssteilnehmer hat auf diesem Mail-Server **postc1** eine sog. *Mail-Box* (Briefkasten), in der die an ihn adressierte und angekommene E-Mail abgelegt wird.

Das Lesen und Verschicken von E-Mail geschieht auf unseren Praktikumsrechnern am besten ebenfalls durch den *Netscape-Navigator*.

Näheres hierzu im ersten Praktikumsstermin!

1.4 Programmerstellung

1.4.1 Prinzipielles zur Programmerstellung

Die ersten Programmiersprachen für die ersten brauchbaren Rechenanlagen (≈ 1950) waren sog. *Maschinensprachen*, Sprachen der *ersten Generation*. Die Instruktionen und Daten für den Prozessor wurden als Bitfolgen direkt in den Arbeitspeicher eingegeben. Dies war sehr aufwendig, fehleranfällig und zu jedem Rechner existierte eine eigene Sprache.

Die *zweite Generation* (ab 1955), die sog. *Assemblersprachen*, verfügten über symbolische Namen für Befehle und Makros für gewisse Befehlsfolgen. Diese *Assemblerprogramme* wurden vom *Assembler* in Maschinensprache umgesetzt. Diese Art der Programmierung war und ist immer noch stark maschinenabhängig.

Die *dritte Generation* von Programmiersprachen (ab 1960), die sog. *höheren Programmiersprachen*, verlagerten den Schwerpunkt von der Maschine, auf der gerechnet werden soll, auf das Problem, welches gelöst werden sollte (*Problemorientierte Sprachen*). Hier wird der Lösungsweg nach gewissen grammatischen Regeln (*Syntaxregeln*) in einer für Menschen lesbaren Form aufgeschrieben.

Dieses Quellprogramm wird dann in mehreren Schritten in ein ablauffähiges Maschinenprogramm umgesetzt.

Der prinzipielle Ablauf der Programmerstellung ist in Abbildung 8 veranschaulicht. In einigen Betriebssystemen können mehrere dieser Schritte in einem Arbeitsgang erledigt werden.

1.4.2 C-Programm-Erstellung unter UNIX

C-Quelltexte, auch Quelldateien, Quellprogramme oder Source-Dateien genannt, haben auf UNIX-Rechnern standardmäßig die Endung `.c`.

Zur Erstellung des C-Quelltextes muss ein Editor (Text-Schreib-Programm) verwendet werden. Ein leistungsfähiger Standard-Editor, welcher auf allen Unix-Systemen vorhanden ist, ist der `vi`-Editor. Dieser wird wie folgt aufgerufen:

```
vi program.c<cr>
```

wobei der Name C-Quelldatei beliebig sein kann, aber wie gesagt mit `.c` enden sollte. Der `vi`-Editor wird in Abschnitt 1.5 vorgestellt.

Nach Erstellung eines (hoffentlich fehlerfreien) C-Quelltextes (sein Name sei `program.c`) muss dieser übersetzt und gebunden und das ausführbare Programm im Dateisystem abgelegt werden.

Dies alles geschieht auf unseren Rechnern durch eins der Kommandos:

```
gcc program.c<cr>    oder    cc program.c<cr>
```

(Der `gcc` ist der für viele Plattformen verfügbare GNU-C-Compiler und der `cc` ist der nicht zur UNIX-Grundausrüstung gehörende C-Compiler von SUN.)

Aus dem Quelltext wird (jeweils) ein ausführbares Programmdatei mit dem Namen `a.out` erzeugt und im aktuellen Verzeichnis abgelegt — natürlich nur, wenn der Quell-

text fehlerfrei war — ansonsten meldet der Compiler derartige Fehler!

Beim Compilieren können zwischenzeitlich mehrere temporäre Dateien erzeugt worden sein, welche aber nicht im Dateisystem stehen bleiben!

Dass diese Datei `a.out` ausführbar ist, erkennt man an deren Rechtevektor oder durch Anwendung des `file`-Kommandos!

Das fertig übersetzte Programm kann dann einfach durch Angabe des Namens gestartet werden:

```
a.out<cr>
```

es wird dann vom System zusammen mit einer Laufzeitumgebung in den Arbeitsspeicher geladen und zur Ausführung gebracht.

Dem ausführbaren Programm kann mittels des `mv`-Kommandos ein anderer Name gegeben werden:

```
mv a.out neuer_name<cr>
```

oder beim Compileraufruf kann durch die `-o`-Option dafür gesorgt werden, dass die ausführbare Programmdatei direkt einen anderen Namen als `a.out` erhält:

```
gcc -o neuer_name program.c<cr> oder cc -o neuer_name program.c<cr>
```

Später werden wir sehen, wie man die einzelnen Schritte des Compilierens einzeln durchführen kann und wie man sich die dabei entstehenden Zwischenergebnisse ansehen kann.

1.4.3 Geschichtliche Entwicklung von C

Die Programmiersprache C wurde im Jahr 1972 von Dennis Ritchie entworfen und erstmals implementiert.

Die anfängliche Weiterentwicklung von C war stark an die des Betriebssystems UNIX gekoppelt.

Im Jahr 1978 folgte die Veröffentlichung des “Manuals“ [Ker 83]: *The C-Programming Language* von B.W. Kernighan und D.M. Ritchie, welche den De-facto-Standard der damaligen UNIX-Implementierungen für C darstellte (“*Kernighan/Ritchie-C*“).

Im Jahr 1988 folgte eine Überarbeitung des Standards durch das *American National Standards Institute* (“*ANSI-C*“) und ein Jahr später erschien die entsprechende Neuauflage des oben genannten Buches [Ker 88], auch in Deutscher Sprache erschienen [Ker 90].

Dem vorliegenden Kurs ist der *ANSI-Standard* zugrundegelegt.

1.5 Der vi-Editor

Ein Editor ist ein “Schreibmaschinenprogramm“ zum Schreiben von Texten — im Gegensatz zu einer Schreibmaschine erscheint das Geschriebene jedoch nicht auf Papier, sondern zunächst nur auf dem Bildschirm bzw. nach “Abspeicherung“ als Datei im UNIX-Dateibaum.

Editoren dienen zur Erstellung einfacher Texte (d.h. keine Sonderzeichen, keine Formatierung, keine unterschiedlichen Schriftarten!).

Der auf allen UNIX-Systemen vorhandene Standard-Editor ist der **vi**.

1.5.1 Aufruf des vi-Editors

Gestartet wird der Editor durch Eingabe des Kommandos:

```
vi filename <cr>
```

wobei **filename** der Name oder Pfad zu der zu editierenden Datei ist.

Existiert diese Datei noch nicht, so wird sie erzeugt. (Hierzu ist das Schreibrecht für das Verzeichnis erforderlich, in dem die Datei stehen soll!)

Existiert diese Datei bereits, so muss man zumindest über das Leserecht für diese Datei verfügen — will man die Datei jedoch tatsächlich verändern, so muss man natürlich auch das Schreibrecht haben!

In beiden Fällen wird die erste Seite des bisherigen Inhalts der Datei auf dem Bildschirm ausgegeben, der blinkende Cursor steht auf dem ersten Zeichen der ersten Zeile der Datei und in der letzten Zeile des Bildschirms steht der Name der Datei mit Angabe der bisherigen Anzahl der Zeichen und Zeilen der Datei. (Bei einer neuen Datei ist der Inhalt zunächst leer und es wird sofort das Dateende angezeigt: Der Bildschirm wird mit Zeilen gefüllt, welche nur aus einem Zeichen “~“ bestehen. Dies ist die **vi**-Notation für Leerzeilen, welche nicht zur Datei gehören!)

Zu beachten ist, dass beim Editieren einer bereits vorhandenen Datei zunächst die Datei vom Dateisystem in den Arbeitsspeicher kopiert wird und (zunächst) nur diese Kopie im Arbeitsspeicher editiert und ggf. verändert wird. Erst durch explizites Zurückschreiben der Arbeitsspeicherkopie ins Dateisystem (etwa mittels des **vi**-Befehls **:w<cr>**) wird der Inhalt der Datei im Dateisystem verändert! Auch beim Editieren einer neuen Datei wird diese erst durch explizites Abspeichern (**:w<cr>**) im Dateisystem erzeugt — vorher existierte nur die Version im Arbeitsspeicher!

1.5.2 Die Modi des vi

Der **vi**-Editor kann sich in drei verschiedenen Modi befinden:

Der Kommandomodus:

Nach dem Aufruf des Editors befindet sich dieser zunächst im *Kommandomodus*. Im Kommandomodus reagiert der **vi** direkt auf eingetippte Zeichen (werden als Kommandos interpretiert), diese werden nicht auf dem Bildschirm ausgegeben und ein “Abschicken“ mittels **<cr>** ist unnötig, meist sogar falsch!

Eine Editorsitzung kann nur im Kommandomodus beendet werden. Befehle hierzu:

Kommando	Wirkung
ZZ	Beenden mit Abspeichern.
:wq<cr>	Beenden mit Abspeichern.
:q<cr>	Beenden, falls die Datei nicht geändert wurde.
:q!<cr>	Beenden ohne Abspeicherung, Änderungen gehen verloren.

Der Einfüge- oder Ersetzungsmodus:

Ist der Editor im *Einfügemodus* bzw. *Ersetzungsmodus*, so werden die eingetippten Zeichen als Zeichen aufgefasst, die in die Datei einzufügen sind (Einfügemodus) bzw. die die Zeichen der Datei überschreiben sollen (Ersetzungsmodus). Diese so eingetippten Zeichen verändern den auf dem Bildschirm stehenden Inhalt (der Kopie) der bearbeiteten Datei. Man gelangt vom Kommandomodus durch folgende Tasten in den Einfügemodus bzw. Ersetzungsmodus:

Taste	Wirkung
a	Der nachfolgend eingegebene Text wird hinter der augenblicklichen Cursorposition eingefügt.
i	Der nachfolgend eingegebene Text wird vor der augenblicklichen Cursorposition eingefügt.
o	Unterhalb der aktuellen Zeile wird zunächst eine Leerzeile eingefügt und diese Leerzeile durch den nachfolgend eingegebenen Text ersetzt.
O	(Großes Oh!) Oberhalb der aktuellen Zeile wird zunächst eine Leerzeile eingefügt und diese Leerzeile durch den nachfolgend eingegebenen Text ersetzt.
s	Das Zeichen, auf dem der Cursor steht, wird durch den nachfolgend eingegebenen Text ersetzt.
cw	Der Teil des Wortes von der aktuellen Cursorposition bis zum Wortende wird durch den nachfolgend eingegebenen Text ersetzt.

Im Einfüge- bzw. Ersetzungsmodus kann man mit der *Backspace*-Taste die zuletzt eingetippten Zeichen wieder löschen.

Der Einfügemodus bzw. Ersetzungsmodus wird durch die Escape-Taste <esc> beendet, d.h. anschließend ist der Editor wieder im Kommandomodus.

Der ex-Modus::

Der *ex-Modus* dient zur Durchführung einiger typischer Editorfunktionen wie Suchen, Suchen und Ersetzen. Man gelangt durch Eintippen eines Doppelpunktes : vom Kommandomodus in den *ex-Modus*. Der Cursor springt sofort auf die letzte (nicht zur Datei gehörende) Zeile des Bildschirms und hier kann man nun ein beliebiges *ex*-Kommando eingeben. Dieses Kommando wird durch ein <cr> abgeschickt und ausgeführt. Nach Beendigung des *ex*-Kommandos ist der Editor wieder im Kommandomodus und der Cursor wieder innerhalb des Dateiausschnitts des Bildschirms. (Noch nicht vollständig eingegebene *ex*-Kommandos können durch <esc> abgebrochen werden. Auch hiernach ist der Editor wieder im Kommandomodus.)

Die Modi im Überblick:

Kommandomodus und Einfügemodus sind i. Allg. äußerlich (d.h. durch Blick auf den Bildschirm) nicht zu unterscheiden! Im Zweifelsfall kann man ein- oder zweimal die Escape-Taste <esc> drücken und man ist anschließend sicherlich im Kommandomodus! Drückt man im Kommandomodus <esc>, so passiert außer einem Flackern des Bildschirms (und dem Ertönen eines Piepstons) nichts Schädliches.

Durch Eingabe des Befehls `:set showmode<cr>` im Kommandomodus wird der vi so eingestellt, dass in der letzten Bildschirmzeile rechts eine entsprechende Meldung steht, falls sich der Editor im Einfüge- oder Ersetzungsmodus befindet.

(Zurücksetzen mit `:set noshowmode<cr> .`)

Je nach Gerät kann es sein, dass im Kommandomodus die Cursortasten , ,  und  tatsächlich funktionieren!

Im Eingabemodus funktionieren sie in der Regel nicht — da diese Tasten i. Allg. von der Tastatur als Escape-Sequenzen (<esc>-Zeichen, anschließend andere Zeichen) übermittelt werden, und sie somit im Eingabemodus den vi wegen der zentralen Bedeutung der <esc>-Taste durcheinander bringen!

Cursorpositionierungen:

Cursorpositionierungen kann man nur im Kommandomodus (oder über ex-Kommandos) durchführen.

Im Kommandomodus stehen u.a. folgende Kommandos (Tasten) zur Verfügung:

Taste		Wirkung
k oder 	*	Cursor geht ein Zeichen nach oben (vorherige Zeile).
j oder 	*	Cursor geht ein Zeichen nach unten (nächste Zeile).
h oder 	*	Cursor geht ein Zeichen nach links (gleiche Zeile). Gleichwertig kann man hierzu auch die Backspace-Taste oder die Tastenkombination <ctrl> h verwenden.
l oder 	*	Cursor geht ein Zeichen nach rechts (gleiche Zeile). Hierzu kann man auch einfach die Leertaste betätigen.

Taste		Wirkung
<ctrl> b		(<u>b</u> ackward) Bildschirm und Cursor geht eine Seite nach oben.
<ctrl> f		(<u>f</u> oreward) Bildschirm und Cursor geht eine Seite nach unten.
w	*	Cursor geht auf den Anfang des nächsten Wortes.
b	*	Cursor geht auf den Anfang des aktuellen Wortes bzw. des vorherigen Wortes, falls der Cursor bereits an einem Wortanfang stand.
0		Cursor geht an den Anfang der aktuellen Zeile.
\$	*	Cursor geht ans Ende der aktuellen Zeile.

Die in der zweiten Spalte mit einem * gekennzeichneten Kommandos können mit einem Wiederholungsfaktor versehen werden. Dies geschieht, indem man eine natürliche Zahl unmittelbar vor dem Zeichen des Kommandos eingibt. Das entsprechende Kommando wird dann entsprechend oft durchgeführt, Beispiel:

Eingabe: **k** Effekt: Cursor geht eine Zeile höher.

Eingabe: **5k** Effekt: Cursor geht 5 Zeilen höher.

Zur Cursorpositionierung stehen folgende **ex**-Kommandos zur Verfügung (die in folgender Tabelle angegebenen Doppelpunkte überführen den Kommandomodus in den **ex**-Modus und gehören somit strenggenommen selbst nicht zum **ex**-Kommando!):

Taste	Wirkung
:0<cr>	Cursor geht auf das erste Zeichen der ersten Zeile der Datei.
:\$<cr>	Cursor geht auf das erste Zeichen der letzten Zeile der Datei.
:n <cr>	Cursor geht auf das erste Zeichen der n-ten Zeile der Datei. Hierbei muss n eine natürliche Zahl sein. Mit dem ex -Befehl :=<cr> kann man die Nummer der aktuellen Zeile erhalten.
:+n <cr>	Cursor geht n Zeilen vorwärts.
:-n <cr>	Cursor geht n Zeilen rückwärts.

Man kann den **vi** so einstellen, dass er vor jeder Zeile der Datei die (natürlich nicht zum Dateinhalt gehörende) Zeilennummer ausgibt.

Dies geschieht durch den **ex**-Befehl **:set nu<cr> .**

Abstellen kann man diese Numerierung mit **:set nonu<cr> .**

1.5.3 Löschen von Text

Löschen von Teilen des Textes ist aus dem Kommandomodus möglich:

Kommando	Wirkung
x	Zeichen unter dem Cursor wird gelöscht.
dw	Wort(ende) unter dem Cursor wird gelöscht (von Cursorposition an bis zum Ende des Wortes).
dd	Ganze Zeile, auf der der Cursor steht, wird gelöscht.
d\$	Rest der Zeile (von Cursorposition an) wird gelöscht.

Auch hier können die Kommandos jeweils mit einem Wiederholungsfaktor versehen werden: Beispiel:

Eingabe: **dd** Effekt: Lösche eine Zeile.

Eingabe: **15dd** Effekt: Lösche 15 Zeilen.

Eingabe: **dw** Effekt: Lösche eine Wort.

Eingabe: **3dw** Effekt: Lösche 3 Worte.

Mehrere Zeilen kann man auch mit folgendem **ex**-Kommando löschen:

:n,m d<cr>

Hierbei sind i. Allg. n und m Nummern von Zeilen der Datei. Der angegebene Bereich (von Zeile mit Nummer n bis Zeile mit Nummer m, diese beiden Zeilen eingeschlossen) wird aus der Datei entfernt. Hierbei muss die Zeile n vor der Zeile m in der Datei stehen!

Beispiel: **:7,25 d<cr>** , lösche Zeile 7 bis Zeile 25.

Anstelle der "Nummern" sind auch "." für aktuelle Zeile (= Zeile des Cursors) und

“\$“ für letzte Zeile der Datei möglich.

Weiterhin können hier Ausdrücke der Form **-nr** oder **+nr** stehen, wobei **nr** eine natürliche Zahl ist. Hier werden entsprechend viele Zeilen von der aktuellen an rückwärts (bei -) bzw. vorwärts (bei +) gezaehlt, etwa:

Beispiel: `:-5,+7 d<cr>` , lösche die aktuelle, die 5 vorherigen und die 7 nächsten Zeilen.

1.5.4 Suchen, Suchen und Ersetzen

Suchen:

Für die einfache Suche stehen zwei Befehle zur Verfügung:

Suche vorwärts: `/muster <cr>`

Suche rückwärts: `?muster <cr>`

Es handelt sich hierbei auch wiederum um **ex**-Kommandos — durch das “/“ bzw. das “?“ geht der Kommandomodus sofort in den **ex**-Modus über; der Cursor steht auf der untersten Bildschirmzeile und dort kann nun das Suchmuster eingegeben werden. Im Text wird in der entsprechenden Richtung nach der nächsten Zeichenkette gesucht, welche mit dem Muster übereinstimmt, und der Cursor springt auf die gefundene Zeichenkette. Sollte bis zum Dateiende (Vorwärtssuche) bzw. bis zum Dateianfang (Rückwärtssuche) die gesuchte Zeichenkette nicht auftreten, so erscheint in der letzten Zeile eine Meldung: `...not found...` und der Cursor bleibt an seiner ursprünglichen Stelle stehen.

Beispiel: `/hanrath`

Von der aktuellen Cursorposition aus wird in Richtung Dateiende nach der nächsten Zeichenkette **hanrath** gesucht.

Durch Eingabe von **n** im Kommandomodus wird der letzte explizit eingegebene Suchbefehl wiederholt (gleiche Suchrichtung), durch Eingabe von **N** wird der letzte explizit eingegebene Suchbefehl wiederholt, aber in entgegengesetzter Suchrichtung!

Durch `/<cr>` “ (ohne Suchstring) wird der letzte Suchvorgang, aber in Richtung Dateiende wiederholt, durch `?<cr>` “ (ohne Suchstring) wird der letzte Suchvorgang, aber in Richtung Dateianfang wiederholt.

Ein weiterer Suchbefehl wird bei Eingabe des Prozentzeichens “%“ durchgeführt:

Steht der Cursor auf einer der Klammern (, [, {, },] oder), so springt der Cursor nach Eingabe von “%“ auf die korrespondierende Klammer. (Es wird davon ausgegangen, dass zu jeder öffnenden Klammer eine gleichartige schließende Klammer gehört!)

Steht etwa der Cursor auf der schließenden Klammer), so wird die zugehörige öffnende Klammer ((natürlich im Text vorher) gesucht. Steht etwa der Cursor auf der öffnenden Klammer {, so wird die zugehörige schließende Klammer } (natürlich im Text hinterher) gesucht.

Gibt es keine korrespondierende Klammer, so bleibt der Cursor an Ort und Stelle.

Reguläre Ausdrücke in Suchmustern:

Im Suchmuster sind einige Sonderzeichen möglich, welche dann eine Sonderbedeutung haben, etwa:

- `^` vor einem Suchmuster.
Die gesuchte Zeichenkette muss am Anfang einer Zeile stehen
- `$` hinter einem Suchmuster.
Die gesuchte Zeichenkette muss am Ende einer Zeile stehen.
- `\<` vor einem Suchmuster.
Die gesuchte Zeichenkette muss am Anfang eines Wortes stehen.
- `\>` hinter einem Suchmuster.
Die gesuchte Zeichenkette muss am Ende eines Wortes stehen.
- `.` in einem Suchmuster.
Anstelle des Punktes im Suchmuster kann in der gefundenen Zeichenkette im Text jedes beliebige Zeichen stehen.

Es gibt weitere Sonderzeichen, mit denen komplexere sog. *reguläre Ausdrücke* formuliert werden können.

Soll ein Sonderzeichen in einem Suchmuster keine Sonderbedeutung haben, so ist diesem Zeichen ein `\` voranzustellen.

Sucheinstellungen:

Mit folgenden Befehlen können Einstellungen vorgenommen werden, welche sich auf alle folgenden Suchvorgänge beziehen:

Befehl	Wirkung
<code>:set ic<cr></code>	(ignore case) Groß- und Kleinschreibung wird bei der Suche nicht unterschieden.
<code>:set noic<cr></code>	Groß- und Kleinschreibung wird unterschieden (Voreinstellung).
<code>:set mag<cr></code>	(<i>Magic-Option</i>) Reguläre Ausdrücke im Suchmuster sind möglich (Voreinstellung).
<code>:set nomag<cr></code>	Reguläre in Suchmustern sind nicht möglich ("Sonderzeichen" haben keine Sonderbedeutung mehr!).
Befehl	Wirkung
<code>:set ws<cr></code>	(<i>wrapped search, Wrapped-Search-Option</i>) Wird bei einer Vorwärtssuche das Dateiende erreicht, so wird vom Dateianfang aus vorwärts weitergesucht. (Hierbei wird eine Meldung Search wrapped around bottom of buffer in der letzten Bildschirmzeile ausgegeben.) Die Suche endet spätestens an der Cursorsposition zu Beginn des Suchvorgangs. Bei einer Rückwärtssuche wird analog verfahren!
<code>:set nows<cr></code>	Suchvorgänge enden am Dateiende bzw. am Dateianfang (Voreinstellung).

Suchen und Ersetzen

Zum Suchen und Ersetzen lautet der Befehl:

```
:s/suchmuster/ersetzungstext/<cr>
```

Es wird in der aktuellen Zeile von der aktuellen Cursorposition aus bis ans Zeilenende nach der nächsten Zeichenkette gesucht, die mit dem Suchmuster übereinstimmt, und diese wird durch den angegebenen Ersetzungstext ersetzt.

Ist die *Magic-Option* (s.o.) gesetzt, so dürfen im **suchmuster** die üblichen (s.o.) regulären Ausdrücke stehen!

Im Ersetzungstext stehen keine regulären Ausdrücke, d.h. etwaige Sonderzeichen haben im Ersetzungstext keine Sonderbedeutung (außer \ zur Maskierung von /, falls / im Ersetzungstext vorkommen sollte!).

Es sind auch Bereichsangaben für Such- und Ersetzungsbefehle möglich:

```
:n,m s/suchmuster/ersetzungstext/<cr> ,
```

wobei n und m wiederum Nummern von Zeilen, “.” für aktuelle Zeile, “\$“ für die letzte Zeile der Datei oder Ausdrücke etwa der Form +7 oder -5 sein können. Der Such- und Ersetzungsbefehl bezieht sich dann auf jede Zeile des angegebenen Bereiches. (In jeder Zeile wird ggf. die erste mit dem Suchmuster übereinstimmende Zeichenkette ersetzt.)

Zwischen dem den Suchbefehl beendenden / und dem <cr> können die Buchstaben g oder c oder auch beide stehen.

Das Zeichen g für global bewirkt, dass in der aktuellen Zeile bzw. in jeder Zeile des angegebenen Bereichs jede mit dem Suchmuster übereinstimmende Zeichenkette ersetzt wird.

Das Zeichen c bewirkt, dass vor einer Ersetzung vom System nachgefragt wird, ob tatsächlich ersetzt werden soll. Hierbei wird die Zeile, in der Ersetzungen vorgenommen werden könnten, am unteren Bildschirmrand (vorletzte Zeile) ausgegeben, in der letzten Bildschirmzeile ist die zu ersetzende Zeichenkette durch ^...^ markiert und man muss dann y<cr> eingeben, falls diese Zeichenkette ersetzt werden soll, bzw. n<cr> , falls diese Zeichenkette nicht ersetzt werden soll!

Durch das Kommando “&“ wird der letzte Such- und Ersetzungsbefehl wiederholt!

1.5.5 Pufferoperationen

Wie bereits erwähnt wird durch den vi nicht die Originaldatei editiert, sondern die Originaldatei wird beim Editoraufruf in den Arbeitsspeicher kopiert und diese Kopie wird durch den Editor bearbeitet. Erst beim Verlassen des Editors mit ZZ oder :wq<cr> wird die Originaldatei im Dateibaum mit dem Inhalt der bearbeiteten Arbeitsspeicherkopie (diese wird auch *genereller Puffer* genannt!) überschrieben.

Man kann ebenfalls während einer Editorsitzung die Originaldatei mit dem Inhalt dieses generellen Puffers durch den Befehl :w<cr> (write) überschreiben! Der Editor wird hierbei nicht verlassen, d.h. man kann weiter editieren. (Es empfiehlt sich, hin- und wieder dieses Kommando einzugeben, um damit die Änderungen zu “sichern“!) Neben diesem generellen Puffer (Kopie der in Bearbeitung befindlichen Datei) verfügt der vi über weitere Puffer (Zwischenspeicher). Der vi kopiert jeweils das zuletzt glöschte Objekt (Zeichen, Wort oder Zeile, jeweils ein oder mehrere) in einen Puffer.

Diesen Pufferinhalt kann man an beliebiger Stelle (Cursorposition) in den Text der Datei hineinkopieren. Dies erfolgt durch **p** (Einfügen hinter der aktuellen Cursorposition) bzw. **P** (Einfügen vor der aktuellen Cursorposition). Durch das Einfügen in den Text bleibt der Pufferinhalt unverändert!

Beispiel: Lösche 5 Zeilen (die aktuelle und die nächsten 4):

5dd

Nun kann man den Cursor mit den üblichen Positionierungsbefehlen an die gewünschte Stelle im Text bringen. Einfügen des Pufferinhalts (die gelöschten 5 Zeilen) hinter der neuen Cursorposition:

p

Dieses Einfügen des Pufferinhalts kann man an anderer Stelle des Textes wiederholen.

Der Befehl **yw** kopiert das Wort unter dem Cursor in den Puffer, ohne das Wort aus dem Text zu entfernen. Der Pufferinhalt kann dann wiederum mit **p** bzw. **P** an anderer Stelle im Text eingefügt werden. Vor **yw** kann wiederum ein Wiederholungsfaktor angegeben werden — es werden dann entsprechend viele Worte in den Puffer kopiert (ohne, dass sie aus dem Text entfernt werden!).

Analog funktioniert der Befehl **yy** zum Kopieren ganzer Zeilen in den Puffer.

1.5.6 Sonstige nützliche vi-Befehle

Blöcke auf Datei schreiben bzw. von Datei lesen:

Man kann die bearbeitete Datei (genereller Puffer) auf eine andere Datei schreiben, Befehle hierzu:

:w dateiname<cr> bzw. **:w! dateiname<cr>**

In der ersten Form ist der angegebene Dateiname der Name (Pfad zu) einer neuen Datei, welche durch diesen Befehl mit einer Kopie des generellen Puffers als Inhalt erzeugt wird (Schreibrecht für das Zielverzeichnis erforderlich!).

In der zweiten Form ist der Dateiname der Name (Pfad zu) einer existierenden Datei, die nun mit dem Inhalt des generellen Puffers überschrieben wird (Schreibrecht für diese Datei erforderlich!).

Soll der Inhalt des generellen Puffers an eine bereits existierende Datei angehängt werden, so ist dies mit dem Befehl:

:w>> dateiname<cr>

möglich.

Schreibbefehle können auch mit Bereichsangaben versehen werden:

:n,m w dateiname<cr> bzw. **:n,m w! dateiname<cr>**

Es wird dann nur der angegebene Bereich von Zeile *n* bis Zeile *m* auf die Datei geschrieben. (*n* und *m* sind hierbei wiederum Zeilennummern, “.” für die aktuelle Zeile oder “\$“ für Dateiende!)

Durch **:r dateiname<cr>** wird der Inhalt der angegebenen Datei (Leserecht erfor-

derlich) hinter die aktuelle Zeile im Text eingefügt.

Ausführung von System-Kommandos:

Man kann in einer vi-Sitzung beliebige Befehle ans System eingeben:

```
:! command<cr>
```

Es wird der UNIX-Befehl `command` ausgeführt, dessen Ausgabe erscheint auf dem Bildschirm (kein Einfluss auf den Inhalt der bearbeiteten Datei). Anschließend wird man aufgefordert, ein `<cr>` einzugeben. Hiernach wird der Bildschirm wieder in der für den vi üblichen Form aufgebaut und man kann weiter editieren.

Als Befehl kann auch der Name einer Shell (etwa `ksh`) eingegeben werden, und es wird (aus dem Editor heraus) ein neuer Kommandointerpreter gestartet, man kann beliebige UNIX-Befehle eingeben und erst nach Verlassen der Shell (`exit`) ist man wieder im Editor.

Durch die folgende Form des Befehls:

```
:r! command<cr>
```

wird wie oben der angegebene UNIX-Befehl ausgeführt, dessen Ausgabe erscheint jedoch nicht nur auf dem Bildschirm, sondern wird hinter der aktuellen Zeile in den Text der bearbeiteten Datei eingefügt!

Umwandlung von Groß- und Kleinbuchstaben:

Ist das Zeichen unter dem Cursor ein Buchstabe, so wird durch Eingabe des Zeichens “~“ der Buchstabe von Groß- in Kleinschreibung umgewandelt bzw. umgekehrt!

Wiederholung des letzten Befehls:

Durch “.” wird der letzte Befehl wiederholt.

Rückgängigmachung von Befehlen:

Durch “u” (undo) wird der letzte Befehl rückgängig gemacht!

Durch “U” werden alle Änderungen der aktuellen Zeile rückgängig gemacht! (Die Zeile darf zwischenzeitlich nicht verlassen werden!)

Aneinanderhängung von Zeilen:

Der Befehl “J” hängt an die aktuelle Zeile die nächste Zeile an (Zeilenvorschubzeichen wird gelöscht!). Vor dem J kann wiederum ein Wiederholungsfaktor mit der üblichen Wirkung angegeben werden, der Befehl “5J” hängt die nächsten 5 Zeilen an die aktuelle Zeile an (die nächsten 5 Zeilenvorschübe werden gelöscht!).

Schaubild	Erläuterung
	<u>CPU</u> (<u>C</u>entral <u>P</u>rocessor <u>U</u>nit), besteht aus Rechen- und Steuerwerk, holt Befehle (Instruktionen) und Daten aus dem Arbeitsspeicher (oder anderen Komponenten), führt die Befehle aus und schreibt veränderte oder neue Daten in den Arbeitsspeicher (oder andere Komponenten). <u>Arbeitsspeicher</u>, auch RAM oder Primärspeicher genannt, lineare Anordnung von Speicherstellen (Bit's), welche den Wert 0 oder 1 annehmen können. In ihm sind die Befehle und Daten der ablaufenden Programme abgespeichert. Speicher ist "flüchtig", d.h. beim Ausschalten des Rechners geht der Inhalt verloren. <u>Terminal-Controller</u>, Schnittstelle zwischen dem Rechner auf der einen Seite und Bildschirm bzw. Tastatur auf der anderen Seite. <u>Festplatten-Controller</u>, Schnittstelle zwischen dem Rechner und den Festplatten oder auch Diskettenlaufwerken (allg. Sekundärspeicher). Auf Sekundärspeichern sind ablauffähige Programme, Daten und Bibliotheken dauerhaft (permanent, auch nach Abschalten des Rechners vorhanden) abgespeichert. <u>Netzwerk-Schnittstelle</u>, Verbindung zu anderen Rechnern (weltweit) zum Austausch von Daten. <u>System-Bus</u> oder kurz: <u>Bus</u>, Verbindung zwischen CPU, Arbeitsspeicher und weiteren Hardware-Schnittstellen. Auf ihm gelangen Daten (und Instruktionen) von einer Komponenten zur anderen. Mögliche weitere Hardware-Schnittstellen zu: Drucker, CD-Rom-Laufwerk, Sound-Karte, Bandlaufwerk ...

Abbildung 1: Modell eines Rechners

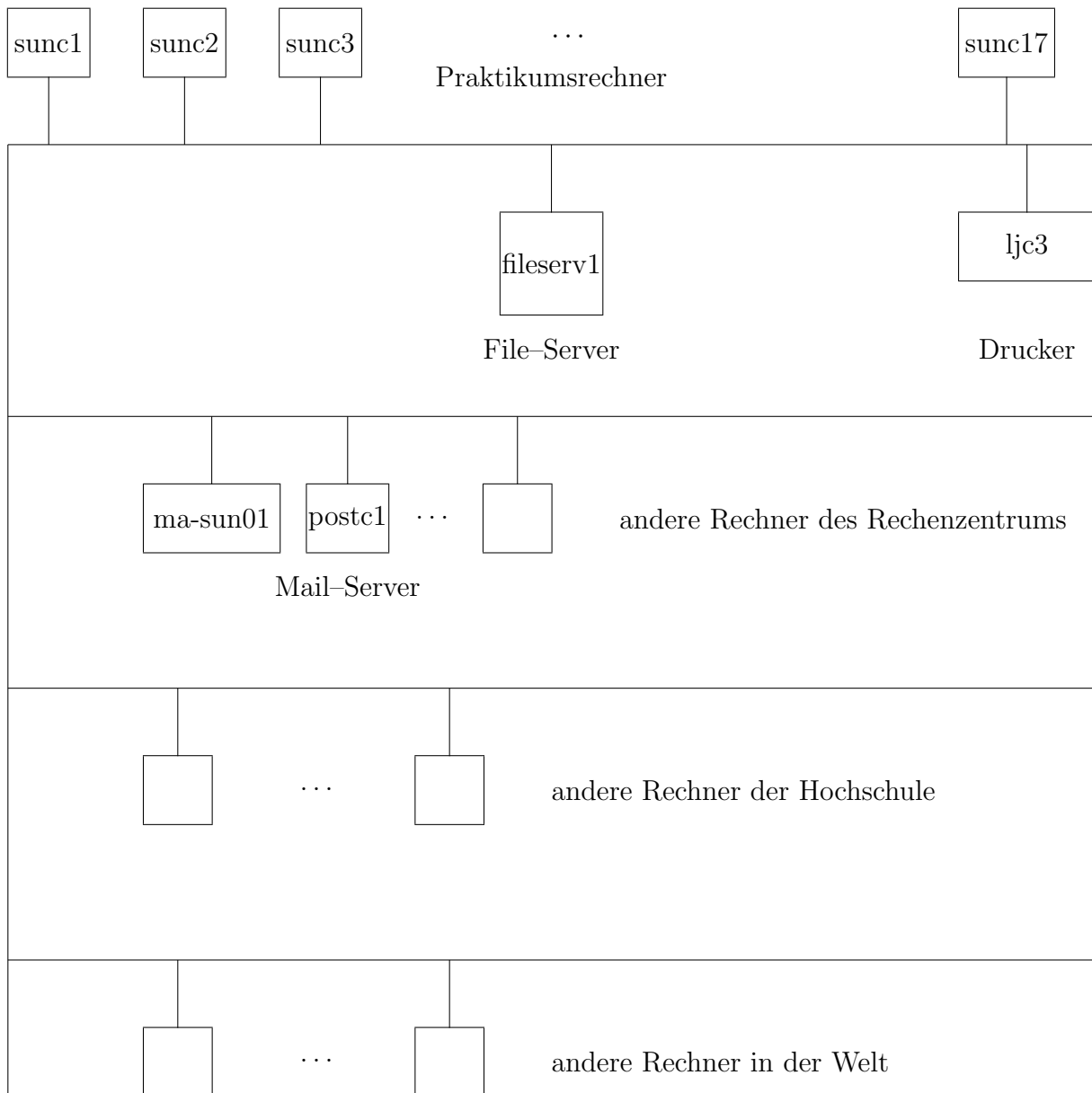


Abbildung 2: Unsere Rechnerlandschaft

Rechenzentrum	Deutschland
?	?
sunc1.rz.rwth-aachen.de	
6	6
Rechner-	Hochschule
name	Aachen

Abbildung 3: Internetname

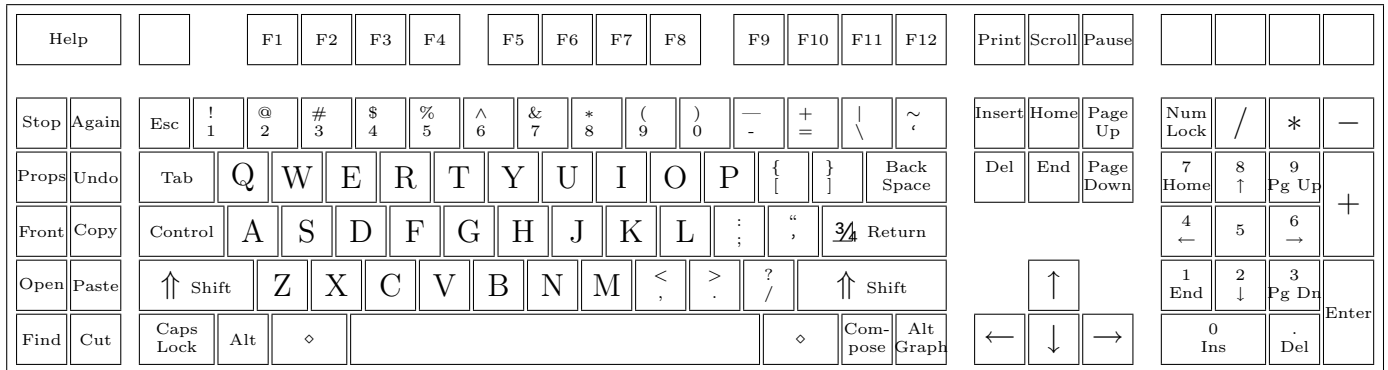


Abbildung 4: SUN-Tastatur

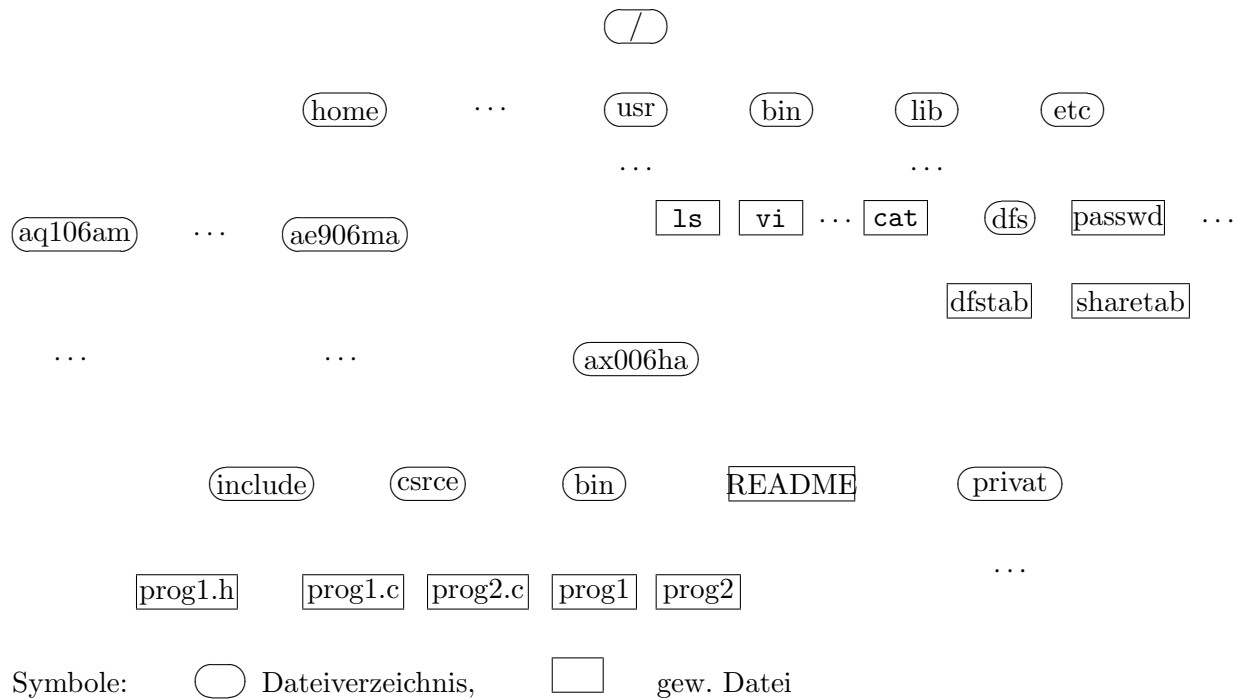


Abbildung 5: UNIX-Dateibaum

Die typische Ausgabe eines `ls -l`-Kommandos sieht wie folgt aus:

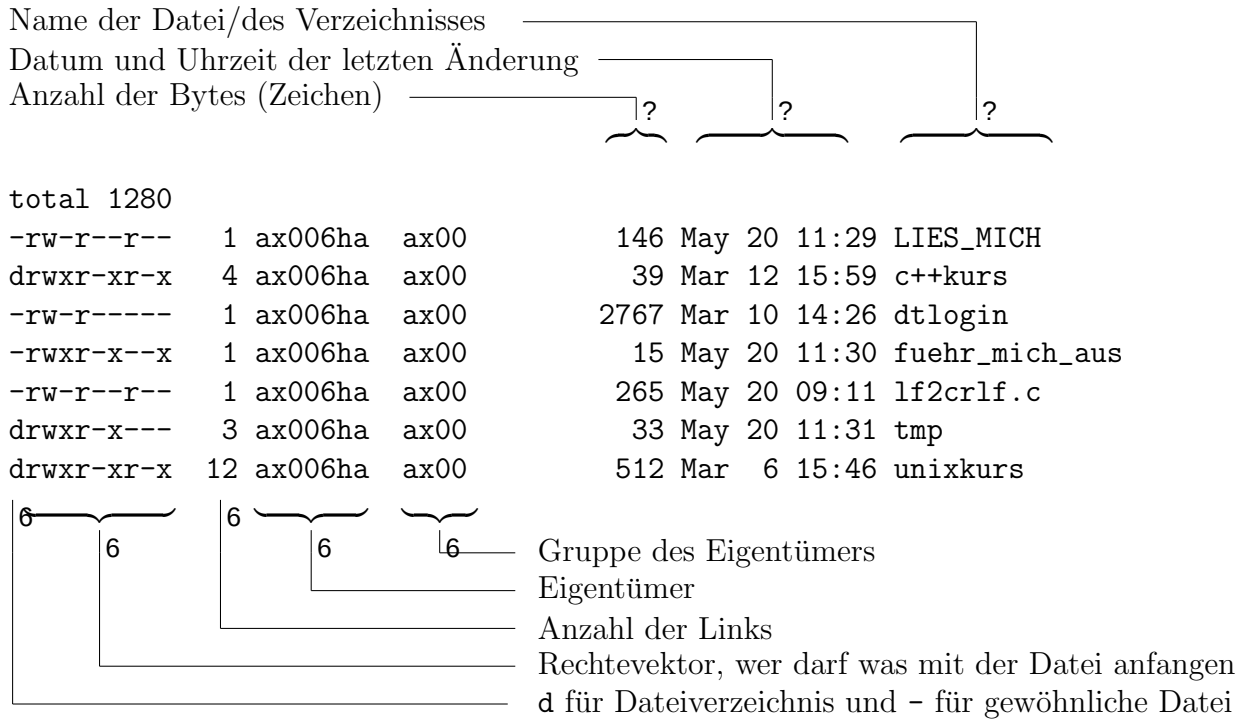
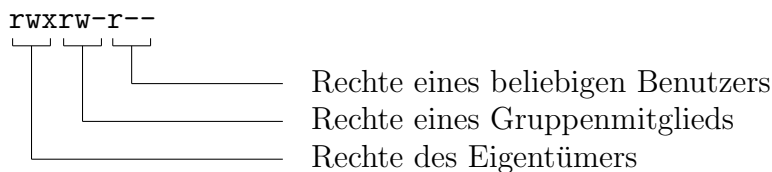


Abbildung 6: Ausgabe des `ls -l`-Kommandos

Ein typischer Rechtevektor ist aus Sicht des Dateieigentümers in drei Abschnitte geteilt:



Jeder derartige Abschnitt eines Rechtevektors ist wiederum dreigeteilt:

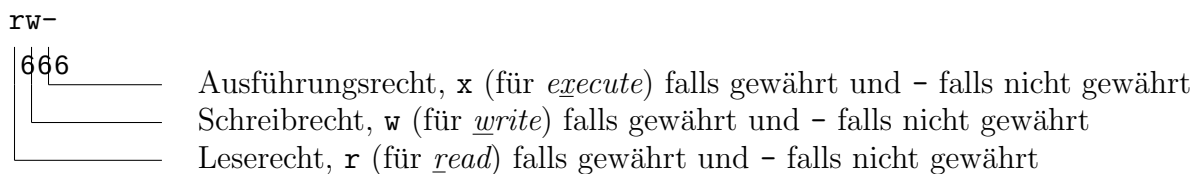


Abbildung 7: Erläuterung zum Rechtevektor

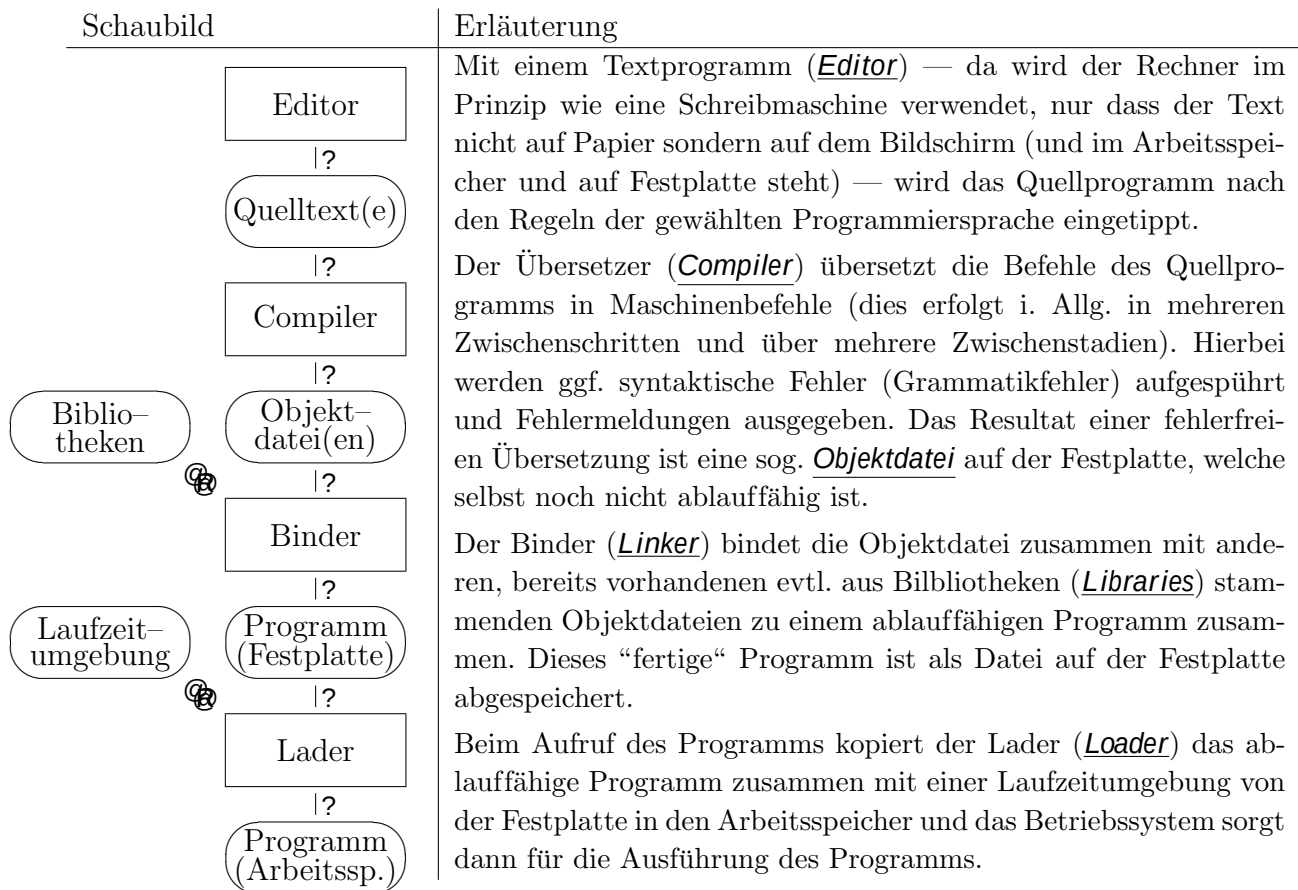


Abbildung 8: Programmerstellung

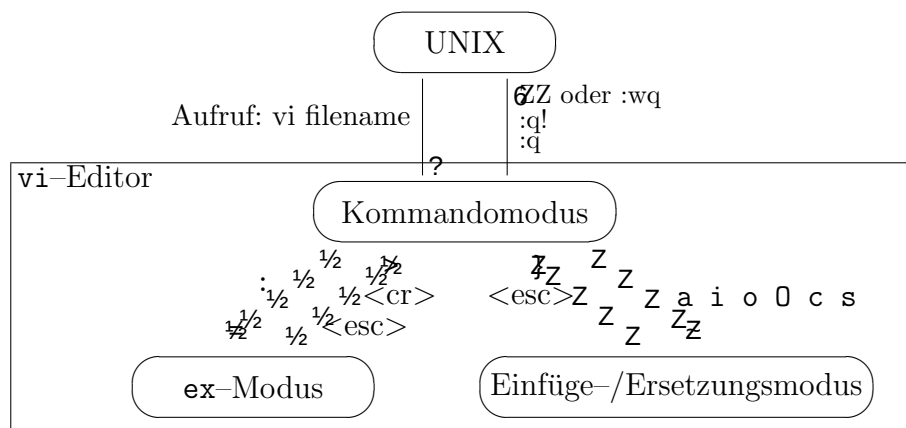


Abbildung 9: Modi des vi-Editors

2 Grundlagen zur C-Programmierung

2.1 Einfachste C-Programme

Ablauffähige Programme liegen im Arbeitsspeicher und bestehen aus dem Anweisungsteil (welche Instruktionen sind durchzuführen) und dem Datenteil (welche Daten werden durch die Instruktionen manipuliert).

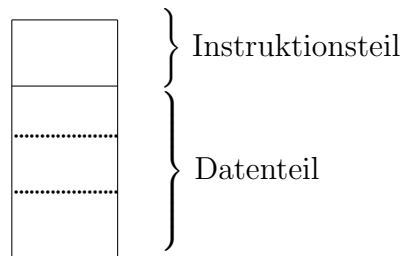


Abbildung 10: Programmaufbau im Arbeitsspeicher

Der Datenteil ist in drei weitere Abschnitte unterteilt, auf deren Bedeutung hier nicht näher eingegangen wird (siehe Kapitel 9).

Ein einfaches C-Programm spiegelt diese Aufteilung (wenn auch in umgekehrter Reihenfolge) in etwa wieder (wobei einige syntaktische Regeln eingehalten werden müssen und es noch ein weiter Weg ist vom C-Quell-Programm bis zum ausführbaren Programm in Maschinensprache im Arbeitsspeicher!):

Beispiel: `einleit/beisp1.c`

(Alle Beispiele sind unterhalb des Verzeichnisses `/home/ax006ha/c_anf` im Dateibaum der Praktikumsrechner zu finden!)

<code>void main(void)</code>	] Programmkopf , so oder so ähnlich syntaktisch vorgeschrieben
{	
<code>int i, j;</code>	] Datendefinitionen
<code>i = 5;</code>	] Anweisungen, Instruktionen
<code>j = i;</code>	
<code>j = j + 17;</code>	
<code>return;</code>	] Ende des Programms, syntaktisch vorgeschrieben
}	

Abbildung 11: Prinzipieller Aufbau eines C-Programms

Die Zeile `void main(void)` heißt Kopfzeile des Programms. Jedes C-Programm muss genau eine derartige Zeile (anstelle des zweimal auftretenden `void` können hierbei andere Dinge stehen, vgl. Abschnitt 7.1).

Hinter der Kopfzeile muss die öffnende geschweifte Klammer stehen, dahinter folgen

die Definitionen von Daten: `int i, j; .`

Hier werden zwei *Variablen* mit den Namen `i` und `j`, jeweils vom *Typ* `int` definiert. Hierdurch werden zwei Speicherbereiche vereinbart, in denen ganzzahlige Werte (Typ `int`) abgespeichert werden können. Diese Speicherbereiche werden im Programm mit den Namen `i` und `j` angesprochen.

Variablen werden zu “Beginn eines Programms“ (hinter der öffnenden geschweiften Klammer) vereinbart — dies geschieht durch die Angabe eines Types durch das entsprechende *Schlüsselwort* (hier `int` für ganzzahlig) und anschließend einer durch Kommata getrennten Liste von *Bezeichnungen* (Namen, dürfen keine *Schlüsselwörter* sein!) und abschließendem Strichpunkt.

Die Vereinbarung von Variablen kann sich auf mehrere Zeilen erstrecken, es können Variablen von unterschiedlichem Type vereinbart werden (siehe Kapitel 3).

Hinter der Vereinbarung von Variablen folgen dann die *Anweisungen* (Instruktionen) — hier zunächst die Anweisung: `i = 5;`, die Variable `i` erhält den Wert 5.

Die Variable (reservierter Speicherbereich) wird durch ihren Namen `i` angesprochen — 5 ist eine sog. *Konstante*, hier augenscheinlich eine *ganzzahlige Konstante* (Typ `int`) und die Variable bekommt den Wert dieser Konstanten zugewiesen.

Das Gleichheitszeichen “=” ist also nicht im streng mathematischen Sinn zu verstehen, sondern bedeutet in C: “wird zu“. Hierbei muss das, was links vom Gleichheitszeichen steht, etwas sein, was einen reservierten Speicherbereich identifiziert (*l-value*, also meistens eine Variable), und rechts vom Gleichheitszeichen muss etwas stehen, was einen gewissen Wert (des zum linksseitig stehenden Speicherbereich passenden Types) hat.

Jede derartige Anweisung ist durch einen Strichpunkt abzuschließen.

Die nächsten beiden Anweisungen sind von der gleichen Art. In der ersten `j = i;` erhält die Variable `j` den gleichen Wert wie `i`, also 5 — in der zweiten `j = j + 17;` wird der Wert der Variablen `j` um 17 erhöht. Auf der rechten Seite dieser Zuweisung steht hier ein sog. *Ausdruck*, das ist eine Verknüpfung von Variablen und Konstanten oder auch anderen Ausdrücken mit *Operatoren* (hier der Plus-Operator für die Addition). Der Ausdruck wird “ausgewertet“, d.h. die Operation wird ausgeführt, und das Ergebnis der Operation wird hier der Variablen links des Gleichheitszeichens zugewiesen. Dass diese Variable selbst im Ausdruck auf der rechten Seite des Gleichheitszeichens vorkommt, spielt hierbei keine Rolle!

Die vorletzte Zeile des Programms: `return;` ist ebenfalls eine Anweisung — durch diese wird der Programmlauf beendet und zum Aufrufer des Programms (hier also das UNIX-Betriebssystem) “zurückgekehrt“. Die schließende geschweifte Klammer am Ende des Programms korrespondiert zu der öffnenden zu Anfang und ist syntaktisch vorgeschrieben. (Die `return`-Anweisung ist in diesem Beispiel strenggenommen nicht unbedingt notwendig — die schließende geschweifte Klammer hat selbst auch schon die entsprechende Wirkung: “Rückkehr“ zum Aufrufer!)

Die Anweisungen eines C-Programms werden von oben nach unten der Reihe nach durchgeführt. Derartige aufeinanderfolgende Anweisungen macht man in Struktogrammen wie in Abbildung 12 kenntlich. Der Programm “fließt“ hier von oben nach

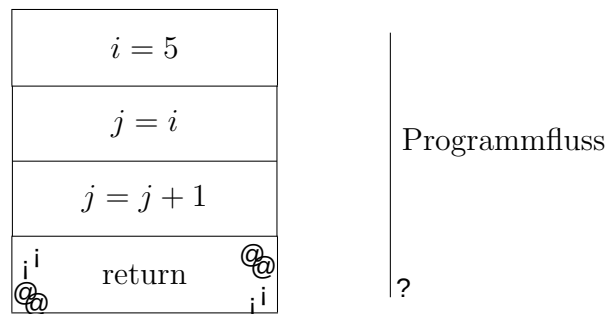


Abbildung 12: Programmfluss

unten. Das unterste Element des Struktogramms ist das Sinnbild für “Rückkehr” zur Aufrufer.

Erstellt man aus diesem Quelltext ein ablauffähiges Programm (Compilieren, Binden) und lässt dieses ablaufen, so meldet sich nach (sehr) kurzer Zeit das System wieder mit seinem Prompt und sonst sieht man nichts! Dies entspricht auch ganz dem anfänglich kennengelernten Rechnermodell — die Aktionen des Programms spielen sich alle zwischen der CPU und dem Arbeitsspeicher ab — der Bildschirm (Terminalcontroller) bleibt hiervon unberührt!

Die Kommunikation eines C-Programms mit der “Außenwelt” (Bildschirm, Tastatur, Festplatte, Netzwerk) erfolgt durch bereits fertig übersetzte Programmteile (Funktionen der Standard-Bibliothek), welche vom Compilerhersteller bereit gestellt werden und in C-Programmen verwendet (aufgerufen) werden können.

Diese Funktionen stehen als Funktionsbibliotheken (in übersetzter Form) auf der Festplatte und werden durch den Binder weitgehend automatisch zum eigenen Programm hinzugebunden!

Bei dem Aufruf einer solchen Funktion werden dieser Daten (Argumente) “mitgegeben” — die Funktion führt ihre “Berechnungen” und andere Aktionen mit den übergebenen (und eigenen) Daten aus und “gibt” ggf. bei ihrem Ende Daten an den Aufrufer “zurück” — und anschließend läuft das aufrufende Programm weiter.

Der Aufruf einer Funktion erfolgt im aufrufenden Programm in einer Anweisung (oder einem Ausdruck) durch Angabe des Funktionsnamen, gefolgt von einer runden öffnenden Klammer, anschließend (optional) eine durch Kommata getrennte Liste von Argumenten (Funktionsargumente), und der schließenden runden Klammer.

Für den Aufruf einer Funktion gibt es in Struktogrammen folgendes Element:

Für das Schreiben auf den Bildschirm steht (u.a.) die `printf`-Funktion zur Verfügung. Dieser Funktion wird als Argument das “mitgegeben”, was auf dem Bildschirm ausgegeben werden soll. Dies ist i. Allg. ein Wort oder ein Satz oder ähnliches.

Der Aufruf sieht (als Anweisung) beispielsweise wie folgt aus:

```
printf(" Berechnungen durchgefuehrt!\n");
```

Das Argument dieses Aufrufs: `" Berechnungen durchgefuehrt!\n"` ist eine sog. konstante Zeichenkette — erkennbar an den einschließenden doppelten Hochkommas. Die Funktion `printf` sorgt dafür, dass die übergebene Zeichenkette auf dem Bild-

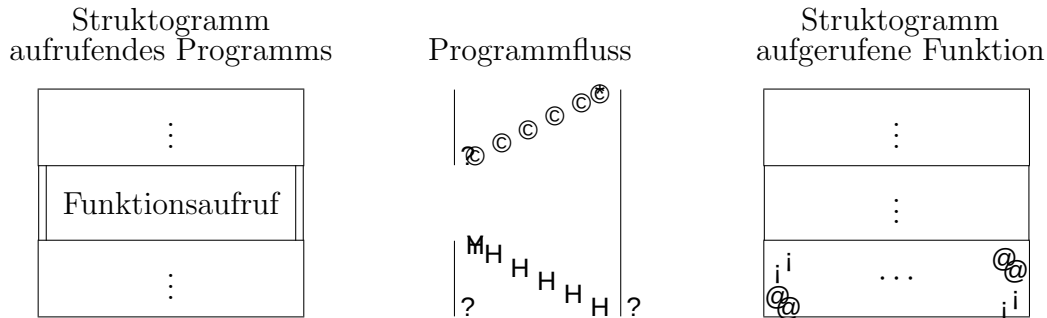


Abbildung 13: Funktionsaufruf

schirm ausgegeben wird. (Die Hochkommas selbst gehören nicht zur Zeichenkette und werden auch nicht ausgegeben!) In konstanten Zeichenketten stehen gewöhnliche Zeichen (Buchstaben, Ziffern, Satzzeichen usw.) und *Sonderzeichen*, hier etwa `\n` für neue Zeile.

Einen Zeilenvorschub bei der Ausgabe kann man in C nur durch “Ausgabe“ des `\n`-Zeichens erhalten! Andere Versuche, etwa

```
printf("Berechnungen durchgefuehrt!")
";
```

funktionieren nicht, sind sogar falsch, da in konstanten C-Zeichenketten kein Zeilenumbruch (explizites, im Editor eingetipptes Zeilenende) stehen darf!

Syntaktisch korrekt ist in diesem Fall das Maskieren des Zeilenendes mit `\` als letztes Zeichen vor dem (im Editor eingetippten) Zeilenumbruch:

```
printf("Berechnungen durchgefuehrt!\n");
```

Das maskierte Zeilenende im Quelltext ist jedoch für den Compiler völlig unerheblich, die beiden beteiligten Zeilen zählen für ihn nur als eine Zeile (Bezeichnung: *Folgezeile*). Zur Erzielung eines Zeilenumbruchs in der Ausgabe müsste auch hier explizit ein `\n` ausgegeben werden:

```
printf("Berechnungen durchgefuehrt!\n\n");
```

Folgezeilen sind überall in C-Quelltexten möglich, etwa auch innerhalb von Bezeichnungen:

```
int ganz\
zahlige_Variable;
```

Hier durch wird eine Variable vom Typ `int` und dem Namen `ganzzahlige_Variable` definiert.

Innerhalb von konstanten Zeichenketten können Folgezeilen vermieden werden, indem man die Zeichenkette in mehrere, eigenständige Zeichenketten aufteilt, etwa:

```
printf("Berechnungen"
" durchgefuehrt!\n");
```

Aneinanderstehende Zeichenketten (dazwischen dürfen Leerzeichen, Tabulatoren oder Zeilenvorschübe stehen) werden von C als eine konstante Zeichenkette aufgefasst!

Neben dem `\n` für Zeilenvorschub gibt es weitere Sonderzeichen: `\t` für Tabulator, `\"` bzw. `\\` falls `"` bzw. `\` selbst Bestandteil der konstanten Zeichenkette sein soll! Obwohl diese Sonderzeichen im Quelltext durch zwei Zeichen dargestellt werden zählen sie nur als ein Zeichen!

Bevor eine Funktion (gilt auch für Bibliotheksfunktionen) verwendet (aufgerufen) werden kann, muss sie dem Compiler "bekannt" gemacht werden, damit er überprüfen kann, ob die Funktion beim Aufruf korrekt verwendet wird (Anzahl und Art der Argumente).

Für die Funktionen der Standardbibliothek geschieht diese Bekanntmachung durch "Einbinden" sog. *Include-Dateien* oder auch *Header-Dateien* genannt. Für die Funktionen der Standard-Ein-Ausgabe ist dies die Header-Datei `stdio.h`.

Die korrekte Verwendung der `printf`-Funktion in unserem Beispiel sieht wie folgt aus:

Beispiel: `einleit/beisp2.c`

<code>#include <stdio.h></code>	] <code>printf</code> u.a. bekanntmachen
<code>void main(void)</code>	] Kopf des Programms
<code>{</code>	
<code>int i,j;</code>	] Datendefinitionen
<code> i = 5;</code>	] Anweisungen
<code> j = i;</code>	
<code> j = j + 17;</code>	
<code> printf(" Berechnungen durchgefuehrt!\n");</code>	] ← Aufruf von <code>printf</code>
<code> return;</code>	] Ende des Programms
<code>}</code>	

Lässt man nun (die übersetzte Version) dieses Programms laufen, so erscheint die im Programm angegebene konstante Zeichenkette auf dem Bildschirm!

Die Bestandteile eines C-Quellprogramms sind:

- *Schlüsselworte* wie etwa `int`. Dies sind Worte mit von C festgelegter Bedeutung.
- *Bezeichner* oder *Identifizier* wie `main`, `printf` oder `i` und `j` in obigem Programm. Dies sind (selbstgewählte) Namen für Funktionen, Variablen usw.. Solche Namen können aus Groß- und Kleinbuchstaben (`[A-Za-z]`), Ziffern (`[0-9]`) und dem Unterstrichungszeichen (`_`) zusammengesetzt sein (dürfen nicht mit einer Ziffer anfangen und nicht mit einem Schlüsselwort übereinstimmen!).
- *Konstanten* wie etwa `5` und `17` (ganzzahlig!) oder auch `3.14159` (Gleitkomma-Konstanten) oder `'a'` (Zeichenkonstanten) oder `" Berechnungen durchgefuehrt!\n"` (konstante Zeichenketten).

- Operatoren wie + für Addition, = für Zuweisung usw..
- Sonstige für die Syntax bedeutsame Zeichen, etwa (und) (Funktionsaufruf), { und } für den *Anweisungsteil* des Programms, ; zur Beendigung von Anweisungen, , zur Trennung von Elementen einer Liste (Definition von Variablen oder Argumente einer Funktion) usw..

Neben diesen elementaren Bestandteilen von C-Quellprogrammen (diese elementaren Bestandteile werden *Token* genannt) gibt es sog. *Präprozessoranweisungen*, etwa `#include <stdio.h>`

Diese bestehen aus ganzen Zeilen und fangen mit dem #-Zeichen an.

Der Präprozessor ist ein Textersetzungsprogramm, welches vom Compiler implizit vor dem eigentlichen Compilieren aufgerufen wird, so dass der “eigentliche“ C-Quelltext erst nach dem Präprozessorlauf vorliegt!

Die Präprozessor `#include`-Anweisung ersetzt die entsprechende Zeile durch den Inhalt der angegebenen Datei!

Zwischen *Token* dürfen im Quelltext “beliebig“ viele (manchmal mindestens eins, etwa in `int i,j`; zur Trennung des Schlüsselwortes `int` vom Bezeichner `i` — sonst würde der Compiler das Token `inti` als Bezeichner auffassen) Leerzeichen, Tabulatoren oder Zeilenvorschübe stehen. Diese sind für den Compiler ansonsten unerheblich.

Für den Compiler wäre auch folgender (hoffentlich abschreckender) Quelltext in Ordnung, er würde genau das gleiche übersetzte Programm liefern:

(Abschreckendes) Beispiel: `einleit/sonicht.c`

```
#include <stdio.h>
void main(void){int i,j;i=5;j=i;j=j+17;printf(
" Berechnungen ausgefuehrt!\n");return;}
```

Zur Erhöhung der Lesbarkeit eines C-Quelltextes sollten einige Konventionen eingehalten werden:

- Pro Zeile nur eine Anweisung.
- Vor und hinter jedem Operationszeichen ein Leerzeichen.
- Anweisungsteil “einrücken“. Geschweifte Klammern, die einen Anweisungsteil umschließen müssen, sollten in der selben Spalte stehen und alle Anweisungen in den Zwischenzeilen sollten einige Positionen weiter rechts anfangen.

Überall zwischen Token darf auch ein *Kommentar* stehen. Kommentare sind für den Compiler unerheblich, sie sollen dem Leser eines Quellprogramms das Verständnis des Programms erleichtern. Kommentare fangen mit `/*` an und hören mit `*/` auf, diese Zeichen und alle dazwischenliegenden Zeichen werden vom Compiler ignoriert. Kommentare können sich über mehrere Zeilen erstrecken, geschachtelte Kommentare sind nicht möglich!

Es folgt die ausreichend kommentierte Version des gleichen Programms:

Beispiel: `einleit/beisp4.c`

```
/******
 * beisp4.c                               *
 *                                         *
 * Beispiel 4 zum C-Kursus                *
 *****/
```

```

*                                     *
* Zweck: Verwendung von Kommentaren *
*                                     *
* Autor: Wilhelm Hanrath            *
* Datum: 18.8.1995                  *
*****/

#include <stdio.h>

void main(void)
{
    /* Variablendefinition */
    int i,j;

    /* Berechnungen
    */
    i = 5;
    j = i;
    j = j + 17;

    /* Ausgabe auf Bildschirm */
    printf(" Berechnungen durchgefuehrt!\n");

    return;
}

```

2.2 Erlaubte Zeichen, Schlüsselworte, Operatoren

2.2.1 Zeichen in C-Quelltexten

Folgende Zeichen dürfen in C-Quelltexten auftreten:

Ziffern:	0...9
Buchstaben:	A...Z a...z
Klammern:	() { } [] < >
sonst. Zeichen:	. , ; ! ? : + - * / = \ ^ ~ % " ' _ # &
	Leerzeichen Tabulator Zeilenvorschub

Es wird zwischen Groß- und Kleinschreibung unterschieden.

2.2.2 Schlüsselwörter

Folgende Schlüsselwörter sind die von ANSI-C vorgeschriebenen. Viele C-Implementierungen haben darüber hinaus noch weitere:

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

Tabelle 2: Schlüsselworte

2.2.3 C-Operatoren

In folgender Tabelle sind die C-Operatoren aufgelistet. Die Bedeutung der einzelnen Operatoren wird später erarbeitet:

Operatoren auf gleicher Höhe haben die gleiche Vorrangstufe (Priorität, etwa wie in der Mathematik: Punktrechnung geht vor Strichrechnung); weiter oben stehende Operatoren haben höhere Priorität als weiter unten stehende.

Zu Operatoren mit gleicher Vorrangstufe ist die sogenannte „Assoziativität“ angegeben, welche festlegt, in welcher Reihenfolge die Auswertung eines Ausdrucks mit gleichrangigen Operatoren erfolgt. (Etwa: beim binären $-$ (Subtraktion) wird der Ausdruck $a-b-c$ in der Reihenfolge $(a-b)-c$ (von links nach rechts) ausgewertet und nicht wie $a-(b-c)$.)

Der Programmierer kann natürlich durch Klammerung (runde Klammern) die Reihenfolge der Auswertung beeinflussen.

Operator	Assoziativität
<code>() [] -> .</code>	von links nach rechts
<code>! ~ ++ -- + - * & (type) sizeof</code>	von rechts nach links
<code>* / %</code>	von links nach rechts
<code>+ -</code>	von links nach rechts
<code><< >></code>	von links nach rechts
<code>< <= > >=</code>	von links nach rechts
<code>== !=</code>	von links nach rechts
<code>&</code>	von links nach rechts
<code>^</code>	von links nach rechts
<code> </code>	von links nach rechts
<code>&&</code>	von links nach rechts
<code> </code>	von links nach rechts
<code>?:</code>	von rechts nach links
<code>= += -= *= /= %= &= ^= = <<= >>=</code>	von rechts nach links
<code>,</code>	von links nach rechts

Tabelle 3: C-Operatoren

Einige Operatorzeichen tauchen mehrfach (zweimal) in der Tabelle auf (z.B. das Minuszeichen $-$ in der zweiten und vierten Zeile von oben!). Dies sind Operatoren, welche in unärer Form (mit einem Operanden, beim Minuszeichen etwa als Vorzeichenminus) und in binärer Form (mit zwei Operanden, beim Minuszeichen etwa bei der Subtraktion) verwendet werden können. Aus dem Kontext, in dem das Operatorzeichen ver-

wendet wird, ist dann (zumindest für den Compiler) immer ersichtlich, ob es sich um die unäre oder binäre Form des Operators handelt.

3 Elementare Datentypen und deren Behandlung

Die Daten eines (übersetzten) Programms (und auch dessen Instruktionen) sind im Arbeitsspeicher abgelegt. Der Arbeitsspeicher ist eine riesige Folge von Speicherstellen (*Bit's*), die jeweils die Information 0 oder 1 aufnehmen können.

Aus organisatorischen Gründen werden je acht derartiger Bit's zu einem *Byte* zusammengefasst und der Arbeitsspeicher kann im Prinzip als eine durchnummerierte Folge von Bytes aufgefasst werden.

Je nach Rechnerarchitektur heißen zwei oder vier oder acht aufeinanderfolgende Bytes ein *Maschinenwort*. Weitere Bezeichnungen:

Name	Abkürzungen	entspricht
Kilobyte	KByte KB	$2^{10} = 1024$ Byte
Megabyte	MByte MB	$2^{10} = 1024$ Kilobyte = 1 048 576 Byte
Gigabyte	GByte GB	$2^{10} = 1024$ Megabyte = 1 073 741 824 Byte
Terabyte	TByte TB	$2^{10} = 1024$ Gigabyte = 1 099 511 627 776 Byte

Abbildung 14: Speichergrößen

Heutzutage sind Arbeitsspeicher auf UNIX-Rechnern bis zu 512 MB groß.

Alle Daten (ganze Zahlen, Gleitkommazahlen, Zeichen und Zeichenketten) müssen im Arbeitsspeicher in Bit's und Byte's, also als Folge von Nullen und Einsen abgespeichert werden. Ob eine Bitfolge nun eine ganze Zahl oder etwa ein Zeichen repräsentiert, ist dieser Bitfolge selbst nicht anzusehen, sondern hängt davon ab, wie der zugehörige Bereich des Arbeitsspeichers interpretiert wird!

Für die Reservierung und Interpretation von Bereichen des Arbeitsspeichers sorgt der Compiler anhand der Definition von Variablen.

Der Typ einer definierten Variablen entscheidet darüber, wieviel Speicherplatz für die Variable reserviert wird, und der Compiler legt fest, wie die Bit-Folge in diesem Speicherbereich im Folgenden interpretiert wird (und welche Operationen mit dieser Variablen möglich sind).

Bei der Definition: `int i`; der Variablen `i` werden beispielsweise auf unseren Praktikumsrechnern vier Byte Speicherplatz reserviert und die Bitfolge in diesen 32 Bit wird immer als ganze Zahl interpretiert.

Bei der Definition: `short int j`; der Variablen `j` werden beispielsweise auf unseren Praktikumsrechnern zwei Byte Speicherplatz reserviert und die Bitfolge in diesen 16 Bit wird immer als ganze Zahl interpretiert.

Es kann also mehrere ganzzahlige Datentypen geben.

3.1 Ganzzahlige Datentypen

Die Darstellung kurzer ganzzahliger Werte (Typ: `short int`) soll für den auf UNIX-Rechnern üblichen Fall (2 Byte) exemplarisch vorgestellt werden. Ist

0	0	1	0	1	1	0	1	1	0	0	1	1	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

die entsprechende sich über 2 Byte (16 Bit's) erstreckende Bitfolge, so ist das rechte Byte das sog. *niederwertige Byte* und das linke das sog. *höherwertige Byte* und die einzelnen Bit's kann man sich von rechts beginnend von 0 bis 15 durchnummeriert denken.

Schreibt man zu jedem Bit die entsprechende Zweierpotenz 2^{Nummer} und "multipliziert" man diese Zweierpotenz mit dem entsprechenden Bit (also mit 0 oder 1), so erhält man die Zahl, die dieses einzelne Bit repräsentiert. Die Summe über alle 16 Bit's ergibt dann den durch diese Bitfolge dargestellten ganzzahligen Wert:

Nummer	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Potenz	2^{15}	2^{14}	2^{13}	2^{12}	2^{11}	2^{10}	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Bitfolge	0	0	1	0	1	1	0	1	1	0	0	1	1	0	0	0

höherwertiges Byte
niederwertiges Byte

Die Bitfolge repräsentiert somit die ganze Zahl:

$$0 \cdot 2^{15} + 0 \cdot 2^{14} + 1 \cdot 2^{13} + 0 \cdot 2^{12} + 1 \cdot 2^{11} + 1 \cdot 2^{10} + 0 \cdot 2^9 + 1 \cdot 2^8 + 1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0$$

$$= 11\,672$$

Auf diese Weise können nur nicht negative Zahlen dargestellt werden, die kleinste ist 0 mit der Bitfolge

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

und die größte ist $\sum_{i=0}^{15} 1 \cdot 2^i = 2^{16} - 1 = 65\,535$ mit der Bitfolge

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

In C existieren mehrere derartige Datentypen zur Darstellung nicht negativer ganzzahliger Werte, welche sich ggf. nur durch die Anzahl der verwendeten Bytes unterscheiden:

Typ-Bezeichnung	Anzahl der Bytes	max. Wert
<code>unsigned char</code>	1	255
<code>unsigned short</code>	mindestens 2	65 535 bei 2 Byte
<code>unsigned</code>	mindestens soviel wie bei <code>unsigned short</code>	4 294 967 295 bei 4 Byte
<code>unsigned long</code>	mindestens 4 und mindestens soviel wie bei <code>unsigned</code>	4 294 967 295 bei 4 Byte

Bei den unteren drei in der Tabelle aufgeführten Typbezeichnungen kann das Wort `int` angehängt werden.

Die genaue Anzahl der verwendeten Bytes ist vom ANSI-Standard nicht weiter festgeschrieben, jeder Compilerhersteller muss sich hier selbst entscheiden.

Zurück zur Zahlendarstellung mit 2 Byte.

Sollen auch negative Zahlen dargestellt werden können, so werden zunächst nur die “rechten” 15 Bit zur Darstellung positiver Zahlen und 0 verwendet. (Die größte so darstellbare positive Zahl ist somit $2^{15} - 1 = 32\,767$.)

Eine negative ganze Zahlen $-n$ (n positiv) wird dann wie folgt dargestellt:

- 1.) Ausgangspunkt ist die beschriebene Darstellung für die positive Zahl n .
- 2.) In dieser Darstellung wird jedes Bit komplementiert. (Aus 0 wird 1 und umgekehrt!)
- 3.) Auf die so erhaltene Darstellung wird binär eine 1 addiert.

Beispiel: Darstellung der negativen Zahl $-11\,672$

1.) Darstellung von 11 672:	<table><tr><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td></tr></table>	0	0	1	0	1	1	0	1	1	0	0	1	1	0	0	0																
0	0	1	0	1	1	0	1	1	0	0	1	1	0	0	0																		
2.) Komplementierung:	<table><tr><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td></tr></table>	1	1	0	1	0	0	1	0	0	1	1	0	0	1	1	1																
1	1	0	1	0	0	1	0	0	1	1	0	0	1	1	1																		
3.) Addition von 1:	<table><tr><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td></tr><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td></tr></table>	1	1	0	1	0	0	1	0	0	1	1	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
1	1	0	1	0	0	1	0	0	1	1	0	0	1	1	1																		
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1																		
Darstellung von $-11\,672$:	<table><tr><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td></tr></table>	1	1	0	1	0	0	1	0	0	1	1	0	1	0	0	0																
1	1	0	1	0	0	1	0	0	1	1	0	1	0	0	0																		

Die Komplementierung und anschließende Addition von 1 heißt *2-er-Komplementierung* — sie kann auf beliebige Bitkombinationen (auch für negative Zahlen) angewendet werden. Die so beschriebene Darstellung positiver und negativer Zahlen und hat folgende Eigenschaften:

- Echte negative Zahlen beginnen mit einem 1-Bit.
- Die Zahlen “0” und “-0” haben die gleiche Darstellung. (Im Gegensatz zur Verwendung von einfachen Vorzeichenbits!)
- Die Subtraktion einer Zahl entspricht der Addition des 2-er-Komplements. (Einfachere Berechnung!)
- Wendet man auf eine negative Zahl das 2-er-Komplement an, so erhält man die entsprechende positive Zahl mit gleichem Betrag.
- Zweimalige Anwendung des 2-er-Komplements lässt einen Wert unverändert.
- Die größte so (mit 2 Byte) darstellbare positive Zahl ist 32 767:

0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

- Die kleinste so (mit 2 Byte) darstellbare negative Zahl ist $-32\,767$:

1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

- Die Bitkombination

1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

kann nicht auftreten!

Wie bei den nicht negativen Datentypen (**unsigned**) existieren in C mehrere Datentypen zur Darstellung positiver und negativer ganzer Zahlen, welche die gleiche Anzahl von Bytes aufweisen wie jeweils die entsprechenden **unsigned**-Typen (den Typbezeichnungen **short** und **long** kann **int** angehängt werden, den Typbezeichnungen **short**, **int** und **long** kann das Wort **signed** vorangestellt werden):

Typ-Bezeichnung	Anzahl der Bytes	min. Wert	max. Wert
signed char	1	-127	127
short	mind. 2	-32 767	+32 767 bei 2 Byte
int	mind. soviel wie bei short	-2 147 483 647	+2 147 483 647 bei 4 Byte
long	mindestens 4 und mind. soviel wie bei int	-2 147 483 647	+2 147 483 647 bei 4 Byte

Die “Umrechnug“ von ganzzahligen Werten in die entsprechende Bitkombination übernimmt der Rechner — der Programmierer kann in Programmen wie gewohnt mit im Dezimalsystem angegebenen ganzen Zahlen arbeiten.

Ist etwa durch **short int i**; eine kurze ganzzahlige Variable **i** definiert (bei dieser Definition reserviert der Compiler auf unseren Praktikumsrechnern wie gesagt 2 Byte für **i**) so wird durch die Zuweisung: **i = 11672** genau die Bitkombination:

0	0	1	0	1	1	0	1	1	0	0	1	1	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

in diesen reservierten Bereich abgespeichert und durch die Zuweisung **i = -11672**; die Bitkombination:

1	1	0	1	0	0	1	0	0	1	1	0	1	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Bisweilen ist es nützlich, ganze Zahlen im Programmtext nicht im Dezimalsystem (Basis 10), sondern im Oktalsystem (zur Basis 8) oder im Hexadezimalsystem (Basis 16) anzugeben. Diese beiden Darstellungsarten haben die Eigenschaft, dass die zugehörige Bitkombination “einfacher“ zu berechnen ist.

Beim Oktalsystem werden je drei Bit’s zusammengefasst — drei Bit’s können 8 verschiedene Kombinationen aufweisen, die mit den Ziffern 0 bis 7 durchnummeriert werden:

Bitkomb.	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
Ziffer	0	1	2	3	4	5	6	7

Beim Hexadezimalsystem werden je vier Bit’s zusammengefasst — vier Bit’s können 16 verschiedene Kombinationen aufweisen, die mit den “Ziffern“ 0 bis 9 , *A*, *B*, *C*, *D*, *E*, *F* durchnummeriert werden (die Buchstaben können auch kleine sein!):

Bitkomb.	0 0 0 0	0 0 0 1	0 0 1 0	0 0 1 1	0 1 0 0	0 1 0 1	0 1 1 0	0 1 1 1
Ziffer	0	1	2	3	4	5	6	7

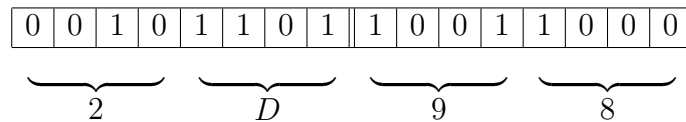
Bitkomb.	1 0 0 0	1 0 0 1	1 0 1 0	1 0 1 1	1 1 0 0	1 1 0 1	1 1 1 0	1 1 1 1
Ziffer	8	9	A	B	C	D	E	F

Die Darstellung der Zahl 11 672 im Oktalsystem lautet somit:

0	0	1	0	1	1	0	1	1	0	0	1	1	0	0	0
2				6				6				3		0	

also 26630. Zur Verdeutlichung, dass es sich um eine Oktalzahl handelt, wird in C-Programmen eine Ziffer 0 vorangehängt, also: 026630.

Die Darstellung der Zahl 11 672 im Hexadezimalsystem lautet somit:



also 2D98. Zur Verdeutlichung, dass es sich um eine Hexadezimalzahl handelt, wird in C-Programmen die "Ziffern" 0X oder 0x vorangehängt, also: 0X2D98.

Die so beschriebene Zuordnung von Oktal- bzw. Hexadezimalzahlen zu den entsprechenden Bitkombinationen bezieht sich nur auf nicht negative Zahlen! Negative Oktal- und Hexadezimalzahlen werden (wie bei Dezimalzahlen üblich) durch ein vorangestelltes Minuszeichen kenntlich gemacht!

So ist z.B. -025530 bzw. -0X2D98 die oktale bzw. hexadezimale Darstellung von -11 672!

Im Folgenden sollen nur die Datentypen `short`, `int` und `long`, jeweils `signed` oder `unsigned`, behandelt werden. Die Behandlung von `char` (`signed` oder `unsigned`) erfolgt im späteren Abschnitt 3.3.

3.1.1 Ganzzahlige Konstanten

Ganzzahlige Konstanten können dezimal, oktal (führende 0) oder hexadezimal (führendes 0X oder 0x), mit oder ohne Vorzeichen (+ oder -) angegeben werden. Diese sind i. Allg. vom Typ `int` (entspricht `signed int`). Ausnahme: ist die dargestellte Konstante größer als der größte durch den Typ `int` darstellbare Wert und ist der durch `long` darstellbare Zahlenbereich größer als der von `int`, so ist die Konstante implizit vom Typ `long` (entspricht `signed long`).

Hängt man ein kleines l oder ein großes L an die (dezimale oder oktale oder hexadezimale) Ziffernfolge der Konstanten an, so ist diese vom Typ `long` (`signed long`).

Hängt man ein kleines u oder ein großes U an die Ziffernfolge an, so ist die Konstante `unsigned` vom entsprechenden Typ (`int` oder `long`).

Konstanten vom Typ `short` (`signed` oder `unsigned`) gibt es nicht!

3.1.2 Definition und Initialisierung ganzzahliger Variablen

Die Definition von Variablen erfolgt, wie bereits gesehen, zu Beginn des Anweisungsteils einer Funktion (Programms) hinter der öffnenden geschweiften Klammer durch Angabe des Types, gefolgt von einer Liste von Bezeichnern und abschließendem Strichpunkt, etwa:

```
...
void main(void)
{
    int i,j;                /* Def. von Variablen vom Typ: int    */
    ...
```

```

long l;           /* dito, Typ: long (= signed long)    */
unsigned u;       /* dito, Typ: unsigned (=unsigned int) */
unsigned short us; /* dito, Typ: unsigned short          */

...
}

```

Hierbei werden jeweils Speicherbereiche entsprechender Größe reserviert — das sind im Augenblick unbenutzte Bereiche des Arbeitsspeichers. Es kann aber sein, dass diese Speicherbereiche vorher bereits schon (vom eigenen Programm oder von einem anderen Programm oder vom Betriebssystem) benutzt wurden und der vorherige Nutzer irgendwelche Bitkombinationen in diesem Speicherbereich “hinterlassen“ hat. Diese (zufälligen) Bitkombinationen werden jetzt als entsprechende ganzzahlige Werte interpretiert, d.h. die so definierten Variablen haben zunächst einen zufälligen Wert! Um mit den Variablen sinnvoll arbeiten zu können, muss der Programmierer ihnen zunächst einen sinnvollen Wert zuweisen (*initialisieren*).

Dies kann durch Zuweisungen im Anweisungsteil des Programms geschehen, bevor die Variablen anderweitig verwendet werden:

```

i = 11675;
j = i + 15;

```

Dies kann aber auch bereits bei der Variablendefinition erfolgen, indem man hinter dem Namen der zu definierenden Variablen ein Gleichheitszeichen und dahinter den Wert schreibt, den die Variable bei ihrer Erzeugung erhalten soll:

```
int i = 11675, j = i + 15;
```

(Hierbei spricht man von *expliziter Initialisierung*. Dies sind formal keine Anweisungen!).

3.1.3 Operationen mit ganzzahligen Werten

Ganzzahlige Werte (Konstanten und Variablen) können mit Operatoren zu Ausdrücken verknüpft werden! Der Ausdruck hat einen Wert (Ergebnis der Operation), welcher im Programm weiterverwendet werden kann.

Die üblichen Operationen für (ganzzahlige) Werte sind:

- Addition (Zeichen: +).
- Subtraktion (Zeichen: -).
- Multiplikation (Zeichen: *).
- Division (Zeichen: /).

Bei der Division ist zu beachten, dass (wie bei Addition, Subtraktion und Multiplikation trivialerweise der Fall) das Ergebnis der Division zweier ganzzahliger Werte ebenfalls ganzzahlig ist — der bei mathematisch korrekt durchgeführter Division entstehende Nachkommateil wird fortgelassen!

Etwa: Die Division $9/7$ liefert das Ergebnis 1 und $5/7$ hat 0 als Ergebnis!

- Ganzzahliger Rest (Zeichen: %, *Modulo-Operator*).

Liefert als Ergebnis den Rest bei ganzzahliger Division.

Etwa `9%7` liefert als Ergebnis 2 (9 geteilt durch 7 ist 1 Rest 2!).

Vorsicht: Ist das Ergebnis einer Operation zu groß für den zugrundeliegenden Datentyp, so können merkwürdige Ergebnisse herauskommen — z.B. die Addition der beiden `short int`-Werte 2 und 32 767 (größter `short int`-Wert bei 2 Byte pro `short int`) liefert die negative Zahl `-32 767` als Ergebnis.

Dieses Phänomen heißt Überlauf oder Overflow.

In Ausdrücken können auch Variablen (Werte) mit unterschiedlichen Typen verknüpft werden, etwa die Addition `s + 1` einer `short`-Variablen `s` und einer `long`-Variablen `1`.

Im Normalfall wird zur Berechnung der Wert des Operanden vom “kleineren“ Typ (also hier der Wert der `short`-Variablen `s`) in den “größeren“ Typen (hier also Typ `long`) umgewandelt (die Variable `s` selbst bleibt hierbei unverändert!) und das Ergebnis der Operation ist dann ebenfalls vom “größeren“ Typ (hier `long`).

In Sonderfällen müssen beide Operanden in einen gemeinsamen “größeren“ Typen umgewandelt werden und der Typ des Ergebnisses ist dann ebenfalls der gemeinsame größere!

Hier taucht ein Problem auf, nämlich, dass es teilweise maschinenabhängig ist, welcher Typ “größer“ als ein anderer ist! Hier etwas präzisere (ANSI-)Regeln:

- Variablen vom Typ `short` werden in (ganzzahligen) Ausdrücken immer zunächst in `int` umgewandelt.

Variablen vom Typ `unsigned short` werden in (ganzzahligen) Ausdrücken in `int` umgewandelt, falls für `int`-Werte mehr Byte reserviert werden als für `short` — ansonsten in `unsigned int`.

Diese standardmäßige Umwandlung von `short`- und `unsigned short`-Werten heißt ganzzahlige Aufwertung.

- `long` ist “größer“ als `int` und `unsigned long` ist “größer“ als `unsigned int`.
- `unsigned int` ist “größer“ als `int` und `unsigned long` ist “größer“ als `long`.
- `long` ist nur dann “größer“ als `unsigned int`, wenn für `long` mehr Byte verwendet werden als für `int`. (Auch hier kann es also — nur auf Maschinen, welche für `long` und `int` gleich viele Byte reservieren — passieren, dass bei der Verknüpfung eines `unsigned int` Wertes mit einem `long`-Wert beide umgewandelt werden, und zwar in `unsigned long`!)

Der Ergebnistyp eines Ausdrucks ist der “größere“ Typ, in den ein oder beide Operanden ggf. umgewandelt wurden!

Möglicherweise überraschende Beispiele:

- Werden auf einer Maschine `short` und `int`-Werte jeweils mit 2 Byte dargestellt und verknüpft (etwa addiert) man einen `unsigned short`-Wert mit einem `int`-Wert, so werden beide zu Berechnung des Ausdrucks in `unsigned int` umgewandelt und das Ergebnis ist ebenfalls vom Typ `unsigned int`!
- Subtrahiert man etwa einen größeren `unsigned int`-Wert von einem kleinerem

`unsigned int`-Wert, so ist das Ergebnis ebenfalls vom Typ `unsigned int` und somit positiv, obwohl mathematisch korrekt ein negatives Ergebnis herauskommen müsste!

Die Zuweisung mit `=` ist strenggenommen ebenfalls ein Operator — dieser hat die Besonderheit, dass der linke Operand ein *l-value* sein muss (keine Konstante, vgl. Seite 40)!

Auch bei Zuweisungen findet ggf. eine Typumwandlung statt: der Wert der rechten Seite der Zuweisung wird in den Typen der linken Seite umgewandelt! Ist hierbei der Typ der linken Seite “kleiner” als der der rechten Seite, so können (müssen nicht!) die Ergebnisse unsinnig sein: ist etwa `s` eine `short`-Variable und `1` eine vom Typ `long`, so macht die Zuweisung `s = 1`; nur dann Sinn, wenn der Wert von `1` größer gleich der kleinsten und kleiner gleich der größten mit `short` darstellbaren Zahl ist!

Die Zuweisung `s = 2147483647`; an eine `short`-Variable `s` ist (höchstwahrscheinlich) unsinnig!

3.1.4 Ausgabe ganzzahliger Werte

In Programmen soll nicht nur mit ganzzahligen Werten gerechnet werden, die Ergebnisse solcher Berechnungen sollen u.U. auch auf den Bildschirm ausgegeben werden. Hierzu kann ebenfalls die `printf`-Funktion der Standardbibliothek verwendet werden. In dem Argument (*Formatstring*, Zeichenkette, die ausgegeben werden soll) zu `printf` können sog. *Ausgabespezifikationen* stehen. Dies sind Zeichenketten, welche mit einem `%`-Zeichen anfangen, optional gefolgt werden von *Ausgabeflaggen* (Steuerzeichen, s.u.) und (nicht optional) mit einem *Umwandlungszeichen* enden.

Zu jeder *Ausgabespezifikation* im Formatstring (es können mehrere vorkommen) korrespondiert ein weiteres Argument zu `printf`. (D.h. innerhalb der runden Klammern zum Funktionsaufruf zu `printf` können mehrere, durch Kommata getrennte Argumente stehen. Das erste Argument ist die Formatzeichenkette und zu “jedem” `%`-Zeichen in der Formatzeichenkette muss hinter der Formatzeichenkette dann ein weiteres Argument, jeweils nach einem Komma folgen!)

Die Umwandlungszeichen legen fest, wie das korrespondierende zusätzliche Argument von `printf` auszugeben ist. Ein Umwandlungszeichen zur dezimalen Ausgabe ganzzahliger Werte ist `d` — zu jeder Umwandlungsspezifikation der Form `%d` im Formatstring muss das korrespondierende Argument ein Ausdruck vom Typ `int` sein und der Wert dieses Ausdrucks wird anstelle des `%d` auf dem Bildschirm ausgegeben.

Beispiel (die Pfeile zeigen jeweils von der Umwandlungsspezifikation auf das zugehörige zusätzliche Argument):

Hat die Variable `i` den Wert 999 und `j` den Wert 15, so wird durch folgenden Aufruf von `printf`:

```

                                ┌──────────┐?
printf("Das Ergebnis von %d plus %d ist %d \n", i, j, i+j );
                                └──────────┘6
                                └──────────┘6

```

folgendes auf dem Bildschirm ausgegeben:

Das Ergebnis von 999 plus 15 ist 1014

(Der Cursor steht anschließend am Anfang der nächsten Zeile!)

Neben `d` als Umwandlungszeichen stehen weitere zur Ausgabe ganzzahliger Werte zur Verfügung:

Zeichen	Bedeutung
<code>i</code>	Wie <code>d</code> , dezimale Ausgabe von <code>int</code> -Werten.
<code>o</code>	oktale Ausgabe von <code>unsigned int</code> -Werten, standardmäßig <u>ohne</u> führende 0.
<code>x</code> bzw. <code>X</code>	hexadezimale Ausgabe von <code>unsigned int</code> -Werten, standardmäßig ohne führendes <code>0x</code> bzw. <code>0X</code> . Bei <code>x</code> werden die Kleinbuchstaben <code>a</code> , ..., <code>f</code> , bei <code>X</code> werden die Großbuchstaben <code>A</code> , ..., <code>F</code> als hexadezimale Ziffern (hinter 9) genommen.
<code>u</code>	Zur dezimalen Ausgabe von <code>unsigned int</code> -Werten.

Gibt man einen `int`-Wert mit einer für `unsigned int` vorgesehenen Umwandlungsspezifikation (`o`, `x`, `X` oder `u`) aus, so wird die Bit-Kombination des auszugebenden `int`-Wertes einfach als `unsigned int` interpretiert, d.h. positive Werte werden unverändert ausgegeben, bei negativen Werten wird das Vorzeichenbit eben nicht als Vorzeichenbit sondern als gewöhnliches Bit interpretiert — das Ergebnis ist dann ein ziemlich großer ganzzahliger Wert!

Bei Ausgabe mit einer derartigen einfachen Umwandlungsspezifikation erscheint anstelle der Spezifikation auf dem Bildschirm der auszugebende Wert mit genau soviel Zeichen dargestellt, wie zu dessen Darstellung notwendig ist. (Für eine dreistellige positive Dezimalzahl werden drei Zeichen ausgegeben — für eine negative Dezimalzahl mit vier Dezimalstellen werden 5 Zeichen, ein Minus für das Vorzeichen und je ein Zeichen für jede Dezimalstelle, ausgegeben.)

Zwischen dem Prozentzeichen und dem Umwandlungszeichen kann eine *Feldbreite* (etwa: `%12d`) angegeben werden. Für jede ausgegebene Zahl erscheinen dann auf dem Bildschirm mindestens sovielen Zeichen, wie in der Feldbreite angegeben wurde (standardmäßig rechtsbündig, ggf. links mit Leerzeichen aufgefüllt!).

Beispiele:

`printf("'%12d'", 1234)`, liefert ! 1234! als Ausgabe (8 Leerzeichen, die beiden Ausrufezeichen im Formatstring und in der Ausgabe dienen hier zur Verdeutlichung der Begrenzung!).

`printf("'%12d'", -123456)`, liefert ! -123456! als Ausgabe (5 Leerzeichen).

Zwischen dem Prozentzeichen und dem Umwandlungszeichen kann eine *Präzision* (etwa: `%.6d`) angegeben werden. In der Ausgabe erscheinen dann für die ausgegebene Zahl mindestens sovielen Ziffern, wie in der Präzision angegeben wurde. (Notfalls werden vorne Nullen ausgegeben!)

Es kann *Feldbreite* und *Präzision* angegeben werden, etwa `%12.6d` mit der naheliegenden Bedeutung: mindestens 12 Zeichen, davon mindestens 6 Ziffern.

Hinter dem Prozentzeichen (vor einer etwaigen Feldbreite oder Präzision) können weiter folgende Steuerzeichen (ggf. kombiniert) angegeben werden:

Steuerzeichen	Bedeutung								
-	Ausgabe erfolgt linksbündig in der angegebenen Feldbreite.								
+	Es wird immer ein Vorzeichen ausgegeben (bei positiven Zahlen das Zeichen +).								
Leerzeichen	bei positiven Zahlen wird anstelle des Vorzeichens ein Leerzeichen ausgegeben.								
0	Es wird mit Nullen bis zur ganzen Feldbreite (Standard: Leerzeichen, s.o.) aufgefüllt.								
#	Alternative Ausgabeform, nur im Zusammenhang mit folgenden Umwandlungszeichen: <table border="1" data-bbox="459 764 1344 930"> <thead> <tr> <th></th><th>Bedeutung</th></tr> </thead> <tbody> <tr> <td>o</td><td>Die oktale Zahl wird mit führender 0 ausgegeben.</td></tr> <tr> <td>x</td><td>Die hexadezimale Zahl wird mit führendem 0x ausgegeben.</td></tr> <tr> <td>X</td><td>Die hexadezimale Zahl wird mit führendem 0X ausgegeben.</td></tr> </tbody> </table>		Bedeutung	o	Die oktale Zahl wird mit führender 0 ausgegeben.	x	Die hexadezimale Zahl wird mit führendem 0x ausgegeben.	X	Die hexadezimale Zahl wird mit führendem 0X ausgegeben.
	Bedeutung								
o	Die oktale Zahl wird mit führender 0 ausgegeben.								
x	Die hexadezimale Zahl wird mit führendem 0x ausgegeben.								
X	Die hexadezimale Zahl wird mit führendem 0X ausgegeben.								

Unmittelbar vor dem Umwandlungszeichen (d, i, o, x, X oder u), hinter einer etwaigen Präzision muss weiterhin ein

- h stehen, falls das zugehörige Argument vom Typ `short` bzw. `unsigned short` ist
- l stehen, falls das zugehörige Argument vom Typ `long` bzw. `unsigned long` ist

3.1.5 Eingabe ganzzahliger Werte

Das Einlesen ganzzahliger Werte während des Programmlaufs von der Tastatur erfolgt mit der zu `printf` analogen Funktion `scanf` aus der Standardbibliothek (`stdio.h` muss "includet" werden!).

Wie `printf` erhält `scanf` beim Aufruf als erstes Argument eine (konstante) Zeichenkette (Formatzeichenkette), in der i. Allg. jedoch nur Eingabespezifikationen vorkommen (wie bei `printf` fangen diese mit einem Prozentzeichen an und enden mit einem Umwandlungszeichen, dazwischen können wieder einige Steuerzeichen auftreten!). Zu jeder Eingabespezifikation gehört dann ein weiteres Argument beim Aufruf von `scanf` und diese weiteren Argumente haben (für uns zunächst) immer die Form: *&Variablenname*, etwa:

```
int i;
...
scanf("%d", &i);
```

Wird die `scanf`-Funktion vom Programm ausgeführt, so hält der eigentliche Programmlauf an und "von der Tastatur" werden solange Zeichen "gelesen", solange diese (beim Umwandlungszeichen d) zur Dezimaldarstellung einer ganzen Zahl gehören:

Es werden führende Leerzeichen, Tabulatoren und `<cr>` ignoriert, es kann dann optional ein Vorzeichen (+ oder -) eingegeben werden, dann eine beliebige Folge von Ziffern. Das erste dann auf der Tastatur eingegebene Zeichen, welches nicht mehr zu einer Dezimalzahl gehören kann (etwa ein Buchstabe oder ein Leerzeichen oder ein `<cr>`) beendet das sog. *Eingabefeld* zu diesem `scanf`-Aufruf — die eingelesenen Ziffern werden als Ziffern einer ganzen Zahl interpretiert und der entsprechende Wert wird der hinter dem `&` angegebenen Variablen (hier: `i`) zugewiesen. Anschließend läuft das Programm weiter.

Im Gegensatz zur Ausgabe auf dem Bildschirm ist das Einlesen von der Tastatur ein komplexerer Vorgang. Entsprechend der Funktionen der Standardbibliothek erfolgt die Eingabe nach folgendem Modell:

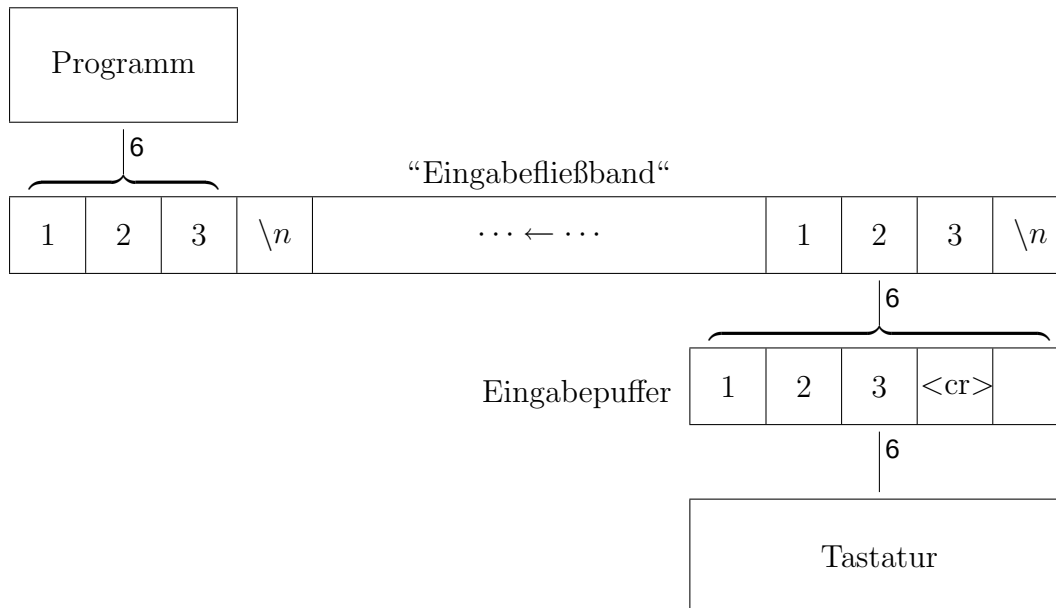


Abbildung 15: Eingabefließband

- Beim Start eines Programms ist das Eingabefließband zunächst leer.
- Auf der Tastatur eingetippte Zeichen stehen zunächst nur im Tastaturpuffer (und auf dem Bildschirm — alles, was eingetippt wird, erscheint auch auf dem Bildschirm!) und noch nicht im Eingabefließband.
Im Tastaturpuffer kann die Eingabe noch mittels der `<bs>`-Taste korrigiert werden.
- Erst durch Tippen von `<cr>` wird der Inhalt des Tastaturpuffers zusammen mit dem Zeichen `\n` für `<cr>` auf das Eingabefließband geschrieben.
- Liegt im Programm ein Eingabebefehl (`scanf` o.ä.) vor, so werden vorne vom Fließband so viele Zeichen gelesen (und dabei vom Fließband entfernt), wie zu der angegebenen Eingabespezifikation “passen“ (bei `%d` etwa anliegende Leerzeichen, Tabulatoren und Zeilenvorschübe (`\n`), optional ein Vorzeichen und eine beliebige Folge von dezimalen Ziffern).

- Ist oder wird beim Einlesen zwischenzeitlich das Eingabefließband leer, so wartet das Programm auf weitere Eingabe, also auf weitere Tastendrucke und anschließendes `<cr>`
- Liegt vorne am Eingabefließband ein Zeichen an, welches nicht mehr zur angegebenen Eingabespezifikation passt (etwa `\n` nach der Ziffernfolge bei der Eingabespezifikation `%d`), so ist der Einlesevorgang mit `scanf` beendet und dieses nicht passende Zeichen bleibt auf dem Eingabefließband stehen und wird ggf. bei späteren Einlesevorgängen vom Eingabefließband genommen.
- Man kann das Eingabefließband durch Eingabe der sog. End-Of-File-Kennung (Abkürzung: *EOF*) “schließen“, d.h. es gelangen anschließend keine neuen Zeichen mehr auf das Fließband. Von einem leeren und geschlossen Eingabefließband kann nicht mehr gelesen werden.
Dieses *EOF* erhält man auf UNIX-Rechnern durch gleichzeitiges Drücken von `<ctrl>` und `d` (auf DOS-Rechnern durch `<ctrl>` und `z`).

Festzuhalten ist, dass jede Eingabe von Tastatur durch `<cr>` “abzuschicken“ ist und dass dieses `<cr>` ein `\n` auf dem Eingabefließband hinterlässt. (Diese `\n`’s auf dem Eingabefließband sind bisweilen lästig, bei der Eingabe ganzzahliger Werte jedoch nicht, da diese beim nächsten Lesen einer ganzen Zahl einfach überlesen und dabei vom Fließband entfernt werden!)

Weiterhin zu unterstreichen ist, dass das Lesen mit dem ersten nicht mehr zur Eingabespezifikation passenden Zeichen endet und dieses Zeichen auf dem Fließband stehen bleibt.

Dies hat folgende Konsequenz:

Sollen in einem C-Programm mehrere ganzzahlige Werte mit `scanf` und `%d` eingelesen werden und tippt man etwa (aus Versehen) als erstes Zeichen einen Buchstaben, so wird beim ersten Einlesen (einer ganzen Zahl) festgestellt, dass dieser Buchstabe nicht mehr zu einer Zahl gehört und das erste Einlesen ist beendet! (Hierbei wurde tatsächlich nichts gelesen und der einzulesenden Variablen auch kein eingelesener Wert zugewiesen! D.h. die Funktion `scanf` ist abgelaufen, konnte aber nicht das von ihr Erwartete leisten!) Dieser nicht passende Buchstabe steht jetzt aber immer noch auf dem Fließband — er wurde ja nicht gelesen — und beim nächsten Lesen mit `scanf` und `%d` ist dieser Buchstabe genauso störend wie beim ersten mal! D.h. solange dieser Buchstabe auf dem Fließband stehen bleibt, können gar keine ganzen Zahlen mehr eingelesen werden!

Mit einem Aufruf der `scanf` können gleich mehrere Werte eingelesen und zugewiesen werden. In der Formatzeichenkette müssen dann mehrere Umwandlungsspezifikationen aufgeführt sein und es müssen entsprechend viele zusätzliche Argumente der Form `&Variablnename` angegeben werden, etwa zum Einlesen und Abspeichern von drei ganzzahligen Werten:

```
scanf("%d%d%d", &i, &j, &k);
```

Zur Kontrolle, ob das Lesen mit `scanf` geklappt hat oder nicht, stellt der Standard folgendes zur Verfügung: Ein Funktionsaufruf von `scanf` hat selbst einen ganzzahligen

gen Wert (Rückgabewert, Funktionsergebnis), nämlich die Anzahl der durch diesen Funktionsaufruf durchgeführten Zuweisungen an Variablen.

Der Aufruf von `scanf("%d", &i)` hat also als Ergebniswert genau dann eine 1, falls der Variablen `i` ein eingelesener Wert zugewiesen werden konnte, und sonst 0 oder ein negativer Wert. Der Aufruf von `scanf("%d%d", &i, &j)` hat also als Ergebniswert genau dann eine 2, falls der Variablen `i` und der Variablen `j` eingelesene Werte zugewiesen werden konnten.

Genau wie der C-Ausdruck `5 + 7` einen Wert hat, nämlich den Wert 12 und dieser Wert des Ausdrucks weiterverwendet, etwa zugewiesen: `j = 5 + 7`, werden kann, hat `scanf("%d" &i)` einen (den oben erläuterten) Wert und dieser Wert kann weiterverwendet, etwa zugewiesen werden:

```
j = scanf("%d", &i);
```

Die Variable `j` (sollte, genau wie `i` vom Typ `int` sein) hat jetzt genau dann den Wert 1, falls das Lesen von `i` geklappt hat! (In Abschnitt 4 werden wir sehen, wie man im C-Programm entsprechend reagiert!)

Eine Möglichkeit zum Lesen (und Entfernen) eines (störenden) Zeichens vorne auf dem Eingabefließband bietet die ebenfalls von der Standardbibliothek zur Verfügung gestellte Funktion

```
int getchar(void);
```

die durch `getchar();` oder `j = getchar();` aufgerufen werden kann. Sie liest und entfernt das nächste auf dem Eingabefließband vorliegende Zeichen. In der zweiten Aufrufform wird der Wert des gelesenen Zeichens der Variablen `j` zugewiesen. Hierzu später mehr!

Neben dem Umwandlungszeichen `d` gibt es für einzulesende `int`-Variablen folgende weitere:

	Bedeutung
<code>o</code>	Die einzulesende Zahl muss oktal, mit oder ohne führende 0 angegeben sein.
<code>x</code>	Die einzulesende Zahl muss hexadezimal, mit oder ohne führendes <code>0x</code> oder <code>0X</code> angegeben sein.
<code>i</code>	die einzulesende Zahl kann dezimal, oktal oder auch hexadezimal angegeben sein. Bei oktalen Zahlen muss eine führende 0, bei hexadezimalen Zahlen ein führendes <code>0x</code> oder <code>0X</code> angegeben sein!

Zur Eingabe von Variablen vom Typ `unsigned` (entspricht `unsigned int`) ist das Umwandlungszeichen `u` zu verwenden.

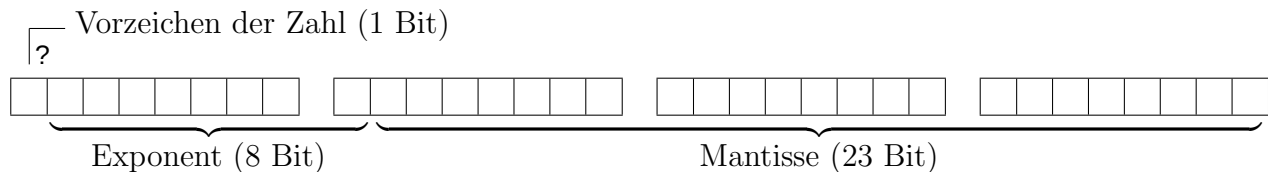
Zur Eingabe von Variablen vom Typ `short` (`signed` oder `unsigned`) ist dem Umwandlungszeichen ein `h` voranzustellen.

Zur Eingabe von Variablen vom Typ `long` (`signed` oder `unsigned`) ist dem Umwandlungszeichen ein `l` voranzustellen.

Zwischen dem Prozentzeichen und dem Umwandlungszeichen der Eingabespezifikation kann wiederum eine Feldbreite angegeben werden, etwa `scanf("%3d", &i)`, es werden dann (nach führenden Leerzeichen, Tabulatoren und Zeilenvorschüben) höchstens

Ziffer vor dem Dezimalpunkt immer eine 1 ist, braucht diese nicht gespeichert zu werden.

Zur Darstellung eines einfach genauen Gleitpunktwertes (Typ: `float`) stehen auf unseren Praktikumsrechnern insgesamt 4 Byte zur Verfügung. Diese sind wie folgt aufgeteilt:



Für den Exponenten wird das Vorzeichen nicht explizit abgespeichert, sondern der Exponent liegt in “verschobener” Form vor, d.h. die Bitkombination

0	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---

repräsentiert den Exponenten 0, die (im Dualsystem) “größeren” Bitkombinationen

repräsentieren positive Exponenten (

1	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

 für Exponent 1,

1	0	0	0	0	0	0	1
---	---	---	---	---	---	---	---

für Exponent 2 usw., der größte so darstellbare Exponent ist

1	1	1	1	1	1	1	0
---	---	---	---	---	---	---	---

 für

Exponent +127), und die (im Dualsystem) “kleineren” Bitkombinationen repräsentieren negative Exponenten (

0	1	1	1	1	1	1	0
---	---	---	---	---	---	---	---

 für Exponent –1,

0	1	1	1	1	1	0	1
---	---	---	---	---	---	---	---

für Exponent –2 usw., der kleinste so darstellbare Exponent ist

0	0	0	0	0	0	0	1
---	---	---	---	---	---	---	---

für Exponent –126). Die Exponenten

0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

 und

1	1	1	1	1	1	1	1
---	---	---	---	---	---	---	---

kommen nicht vor!

Die Bedeutung der Mantissenbits ist ähnlich zu der von ganzzahligen Werten — die

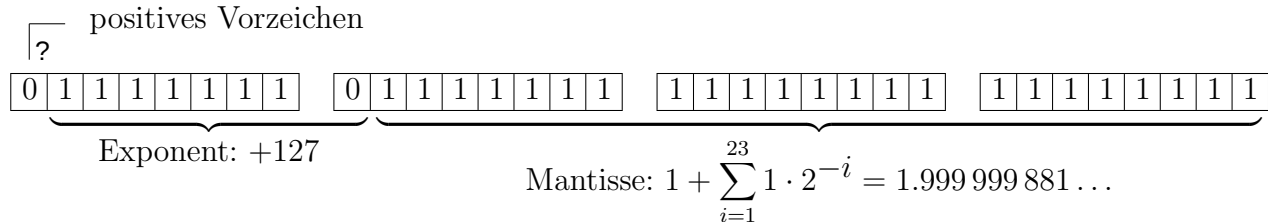
Mantissenbits sind von links mit 1 beginnend durchnummerieren und jedes Bit (0

oder 1) ist mit der entsprechenden 2-er Potenz (negativer Exponent!) $2^{-\text{nummer}}$ zu

multiplizieren. Die Summe über alle Bits und 1 draufaddiert (für die nicht explizit

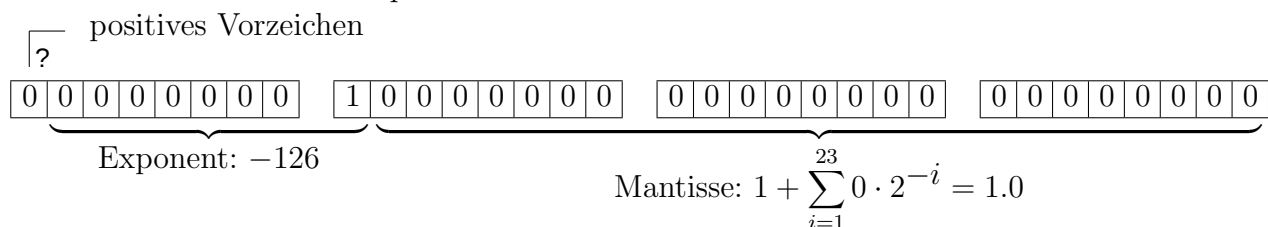
abgespeicherte Ziffer 1 vor dem Dezimalpunkt) ergibt dann den Wert der Mantisse.

Die größte so darstellbare positive Zahl ist



also $+1.999\,999\,881 \dots \cdot 2^{+127} \approx 3.40282346638 \cdot 10^{38}$.

Die kleinste so darstellbare positive Zahl ist



also $+1.0 \cdot 2^{-126} \approx 1.17549435 \cdot 10^{-38}$.

Negative reelle Zahlen werden durch eine 1 im ersten Bit gekennzeichnet.

Bei der Gleitpunktzahl 0.0 sind alle 32 Bit auf 0 gesetzt.

Festzuhalten ist, dass mit dieser Zahlendarstellung nur endlich viele verschiedene positive reelle Zahlen dargestellt werden können und dass diese alle etwa in dem Bereich von 10^{-38} bis 10^{+38} liegen und etwa 6 verlässliche (dezimale) Nachkommastellen haben. Größere bzw. kleinere Zahlen können nicht dargestellt werden und sollte durch Operationen (etwa Multiplikation zweier sehr großer Zahlen) ein Ergebnis entstehen, welches nicht mehr im darstellbaren Zahlenbereich liegt, so entsteht ein *Overflow* bzw. *Underflow*. (Bei einem *Underflow* wird i. Allg. mit 0.0 weitergerechnet, bei einem *Overflow* hört i. Allg. das Programm mit einer Fehlermeldung auf!)

Neben dem vorgestellten Typ `float` zur Darstellung einfach genauer Gleitpunktwerte gibt es in C noch die Typen `double` und `long double`, welche (möglicherweise) mehr als 4 Byte verwenden. (Möglicherweise, weil sich hier wiederum der ANSI-Standard nicht ganz festlegt. Vorgeschrieben ist nur, dass für `double` mindestens soviel Bytes verwendet werden wie für `float` und für `long double` mindestens soviel wie für `double`!)

Auf unseren Praktikumsrechnern werden für `double`-Werte 8 Byte reserviert: 1 Vorzeichenbit, 11 Bits für den (wiederum verschobenen) Exponenten und die restlichen 52 für die Mantisse. Die größte hiermit darstellbare positive Zahl ist $1.797\,693\,134\,862\,315 \cdot 10^{+308}$ und die kleinste $2.225\,073\,858\,507\,201 \cdot 10^{-308}$. Hiermit dargestellte Zahlen weisen etwa 15 verlässliche dezimale Nachkommastellen auf.

Für `long double`-Werte werden auf unseren Praktikumsrechnern 16 Byte reserviert: 1 Vorzeichenbit, 15 Bits für den (wiederum verschobenen) Exponenten und die restlichen 112 für die Mantisse. Der darstellbare (positive) Zahlenbereich ist etwa von 10^{-4932} bis 10^{+4932} und hiermit dargestellte Zahlen weisen etwa 32 verlässliche dezimale Nachkommastellen auf.

3.2.1 Gleitpunktkonstanten

Gleitpunktkonstanten können in C-Quelltexten nur dezimal angegeben werden. Sie bestehen aus dem (optionalen) Vorzeichen, den Vorkommastellen, Dezimalpunkt, dem Nachkommasteil und dem Exponenten. Der Exponent beginnt mit `e` bzw. `E`, dahinter wiederum (optional) ein Vorzeichen und anschließend eine dezimale Ziffernfolge.

Der Vorkommasteil oder der Nachkommasteil kann fehlen — aber nicht beide.

Der Dezimalpunkt oder der Exponent kann fehlen — aber nicht beide.

So angegebene Konstanten sind vom Typ `double`.

Hängt man der Darstellung ein `f` oder `F` an, so ist der Typ `float`. Hängt man ein `l` oder `L` an, so ist der Typ `long double`. Gleitpunktkonstanten brauchen nicht normalisiert zu sein! Beispiele:

	Typ
-1.234	double
1E-3	double
.5e24	double
-1234.5678f	float
.5E24L	long double

3.2.2 Definition von Gleitpunktvariablen

Die Definition von Gleitpunktvariablen erfolgt (wie bei ganzzahligen Variablen) zu Anfang des Anweisungsteils des Programms durch Typangabe, anschließend einer Liste von Namen und durch den abschließenden Strichpunkt.

Bei der Definition kann die definierte Variable gleichzeitig vorbesetzt werden (Gleich-

punktvariable “kleineren“ Types ist auch erlaubt — ist aber nur dann sinnvoll, wenn der zugewiesene Wert im darstellbaren Bereich des “kleinen“ Types liegt.

3.2.4 Ausgabe von Gleitpunktwerten

Zur Ausgabe von Gleitpunktwerten vom Typ `float` oder vom Typ `double` auf den Bildschirm mit der `printf`-Funktion stehen folgende Umwandlungszeichen zur Verfügung:

	Wirkung
f	Exponentenfreie Ausgabe. Bei negativen Zahlen wird zunächst das Vorzeichen ausgegeben, in jedem Fall anschließend der Vorkommateil (mindestens eine Ziffer!), anschließend der Dezimalpunkt und (im Standardfall) 6 Nachkommastellen.
e oder E	Wissenschaftliche Ausgabe mit Exponent. Ggf. das negative Vorzeichen, eine Vorkommastelle, Dezimalpunkt, (im Standardfall) 6 Nachkommastellen, kleines e oder großes E (entsprechend des Umwandlungszeichens), Vorzeichen des Exponenten (auch bei positivem Exponent) und Ziffernfolge des Exponenten.
g oder G	Ist der Exponent des auszugebenden Wertes größer als -4 und kleiner als die Anzahl der auszugebenden Nachkommastellen (im Standardfall 6), so ist die Ausgabe wie bei %f — ansonsten wie bei %e (bei g) bzw. %E (bei G).

Wie bei Ausgabespezifikationen für ganzzahlige Werte kann auch bei Gleitpunktwerten eine Feldbreite (Mindestanzahl der auszugebenden Zeichen) und eine Präzision (Anzahl der Nachkommastellen, im Standardfall 6) angegeben werden.

Hinter dem Prozentzeichen (vor einer etwaigen Feldbreite oder Präzision) können wiederum folgende Steuerzeichen (ggf. kombiniert) angegeben werden:

Steuerzeichen	Bedeutung
-	Ausgabe erfolgt linksbündig in der angegebenen Feldbreite.
+	Es wird immer ein Vorzeichen ausgegeben (bei positiven Zahlen das Zeichen +).
Leerzeichen	bei positiven Zahlen wird anstelle des Vorzeichens ein Leerzeichen ausgegeben.
0	Es wird mit Nullen bis zur ganzen Feldbreite (Standard: Leerzeichen, s.o.) aufgefüllt.
#	Alternative Ausgabeform: es wird immer ein Dezimalpunkt ausgegeben (ansonsten fällt er bei der Ausgabe ganzzahliger Werte ohne Nachkommastellen fort!) und bei g bzw. G werden terminierende Ziffern 0 ausgegeben (standardmäßig werden sie nicht ausgegeben!).

Ist der auszugebende Wert vom Typ `long double`, so muss dem Umwandlungszeichen ein großes **L** vorangestellt werden.

Ein Beispielprogramm zur Ausgabe von Gleitpunktwerten ist: `typen/double/dausg.c`.

3.2.5 Eingabe von Gleitpunktwerten

Zum Einlesen eines Wertes in eine Variable vom Typ `float` stehen die drei (gleichwertigen) Umwandlungsspezifikationen `%e`, `%f` und `%g` zur Verfügung.

Diese Eingabespezifikationen lesen führende Leerzeichen, Tabulatoren und Zeilenvorschübe, ein optionales Vorzeichen, eine Ziffernfolge, welche einen Dezimalpunkt enthalten kann, und einen optionalen Exponenten, bestehend aus `e` oder `E`, einem optionalen Vorzeichen und einer ganzen Zahl. Das erste Zeichen, welches nicht mehr zu dem Gleitpunktwert gehören kann, beendet das Eingabefeld (und dieses Zeichen bleibt auf dem Eingabefließband stehen!).

Die Name der `float`-Variablen, welcher der gelesene Wert zugewiesen werden soll, ist wiederum als weiteres Argument in der Form `&Variablenname` beim Aufruf von `scanf` anzugeben.

Ein `*` hinter dem Prozentzeichen einer Eingabespezifikation `*@*!` in Eingabespezifikation unterbindet wiederum eine Zuweisung und es kann auch wieder eine Feldlänge angegeben werden, welche die Höchstzahl der einzulesenden Zeichen (bis auf führende Leerzeichen etc.) festlegt.

Soll eine `double`-Variable eingelesen werden, so ist dem Umwandlungszeichen ein kleines `l` voranzustellen. (Man beachte den Unterschied: bei der Ausgabe von Gleitpunktwerten mit `printf` wird zwischen `float` und `double`-Werten nicht unterschieden, d.h. es ist bei der Ausgabe von `double`-Werten kein `l` vor dem Umwandlungszeichen notwendig — bei der Eingabe mit `scanf` wird sehr wohl unterschieden und das `l` muss beim Lesen von `double`-Werten dem Umwandlungszeichen vorangestellt werden!)

Soll eine `long double`-Variable eingelesen werden, so ist dem Umwandlungszeichen ein großes `L` voranzustellen.

Wie üblich ist der Ergebniswert eines `scanf`-Aufrufes die Anzahl der Zuweisungen an Variablen.

Gleitpunktwerte können zusammen mit ganzzahligen Werten in einem entsprechenden `scanf`-Aufruf eingelesen werden:

Beispiel:

```
...
void main(void)
{
    int i;
    float x;
    ...
    scanf("%d%g",&i,&x);
    ...
}
```

3.3 Zeichen

Zur Darstellung eines einzelnen Zeichens (Buchstabe, Ziffer, Satzzeichen, Klammer, ...) verwendet jeder C-Compiler ein Byte.

Es können somit $2^8 = 256$ verschiedene Zeichen dargestellt werden.

Zeichen werden in C wie “kleine ganze Zahlen“ behandelt, beim Typen `unsigned char` laufen diese Zahlen von 0 bis 255 — beim Typen `signed char` von -127 bis $+127$. Die Darstellung dieser ganzen Zahlen in einem Byte ist analog zu der vorgestellten Darstellung von (vorzeichenlosen oder vorzeichenbehafteten) ganzzahligen Datentypen. Der Datentyp `char` (ohne vorheriges `signed` oder `unsigned`) ist gleichwertig zu `signed char` oder `unsigned char`, der ANSI-Standard legt sich hier nicht fest.

Die Zuordnung zwischen den Zeichen und den ganzzahligen Werten erfolgt durch den sog. Maschinenzeichensatz — das ist eine Durchnummerierung der auf der Maschine bekannten Zeichen.

Auf heutigen Rechnern hat sich der sog. ASCII-Zeichensatz etabliert. Dieser kennt nur 128 Zeichen mit den Nummern $0, \dots, 127$ und verwendet nur die rechten 7 Bits eines Bytes. (Zur Darstellung dieser Zeichen als `char`-Wert ist es also unerheblich, ob `char` nun `unsigned char` oder `signed char` ist!)

Folgendes Schaubild erläutert die darauf folgende Tabelle dieses 7-Bit-ASCII-Zeichensatzes (Quelle: Dr. Jobst Hoffmann):

b7				0			
b6				1			
b5				0			
b4				b3			
b2				b1			
⋮				⋮			
1				75			
0				113			
1				4B			
⋮				⋮			

Abbildung 16: Dezimaler, oktaler und hexadezimaler Wert eines Zeichens

Die Zeile, in der ein Zeichen steht, legt die Bitkombination der rechten vier Bits (`b1` bis `b4`) und die Spalte die der linken vier Bits (`b5` bis `b7`, das achte Bit `b8` spielt beim 7-Bit-Zeichensatz keine Rolle) des zu einem Zeichen gehörenden Byte fest.

Der Großbuchstabe K wird also durch die Bitkombination

	b8	b7	b6	b5	b4	b3	b2	b1
	1	0	0	1	0	1	1	

 dargestellt. Diese Bitkombination stellt die Dezimalzahl 75 dar (links über dem Zeichen K in der Tabelle abzulesen). Das Zeichen K ist also das 75-te Zeichen des ASCII-Zeichensatzes. Die oktale Darstellung dieser Nummer (hier: 113) ist links unter und die hexadezimale Darstellung (hier: 4B) ist rechts unter dem Zeichen aufgeführt.

b7 b6 b5				0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1								
Bits b4 b3 b2 b1				Steuer- zeichen		Symbole Ziffern		Groß- buchstaben		Klein- buchstaben									
0	0	0	0	0 0	NUL 0	16 20	DLE 10	32 40	SP 20	48 60	0 30	64 100	@ 40	80 120	P 50	96 140	` 60	112 160	p 70
0	0	0	1	1 1	SOH 1	17 21	DC1 11	33 41	! 21	49 61	1 31	65 101	A 41	81 121	Q 51	97 141	a 61	113 161	q 71
0	0	1	0	2 2	STX 2	18 22	DC2 12	34 42	" 22	50 62	2 32	66 102	B 42	82 122	R 52	98 142	b 62	114 162	r 72
0	0	1	1	3 3	ETX 3	19 23	DC3 13	35 43	# 23	51 63	3 33	67 103	C 43	83 123	S 53	99 143	c 63	115 163	s 73
0	1	0	0	4 4	EOT 4	20 24	DC4 14	36 44	\$ 24	52 64	4 34	68 104	D 44	84 124	T 54	100 144	d 64	116 164	t 74
0	1	0	1	5 5	ENQ 5	21 25	NAK 15	37 45	% 25	53 65	5 35	69 105	E 45	85 125	U 55	101 145	e 65	117 165	u 75
0	1	1	0	6 6	ACK 6	22 26	SYN 16	38 46	& 26	54 66	6 36	70 106	F 46	86 126	V 56	102 146	f 66	118 166	v 76
0	1	1	1	7 7	BEL 7	23 27	ETB 17	39 47	' 27	55 67	7 37	71 107	G 47	87 127	W 57	103 147	g 67	119 167	w 77
1	0	0	0	8 10	BS 8	24 30	CAN 18	40 50	(28	56 70	8 38	72 110	H 48	88 130	X 58	104 150	h 68	120 170	x 78
1	0	0	1	9 11	HT 9	25 31	EM 19	41 51	) 29	57 71	9 39	73 111	I 49	89 131	Y 59	105 151	i 69	121 171	y 79
1	0	1	0	10 12	LF A	26 32	SUB 1A	42 52	* 2A	58 72	: 3A	74 112	J 4A	90 132	Z 5A	106 152	j 6A	122 172	z 7A
1	0	1	1	11 13	VT B	27 33	ESC 1B	43 53	+ 2B	59 73	; 3B	75 113	K 4B	91 133	[5B	107 153	k 6B	123 173	{ 7B
1	1	0	0	12 14	FF C	28 34	FS 1C	44 54	, 2C	60 74	< 3C	76 114	L 4C	92 134	\ 5C	108 154	l 6C	124 174	 7C
1	1	0	1	13 15	CR D	29 35	GS 1D	45 55	- 2D	61 75	= 3D	77 115	M 4D	93 135	] 5D	109 155	m 6D	125 175	} 7D
1	1	1	0	14 16	SO E	30 36	RS 1E	46 56	. 2E	62 76	> 3E	78 116	N 4E	94 136	^ 5E	110 156	n 6E	126 176	~ 7E
1	1	1	1	15 17	SI F	31 37	US 1F	47 57	/ 2F	63 77	? 3F	79 117	O 4F	95 137	- 5F	111 157	o 6F	127 177	DEL 7F

Tabelle 4: Zeichensatztafel

Die Steuerzeichen stammen zum Teil noch aus den Anfängen der Datenverarbeitung. Deren Bedeutung ist in folgender Tabelle aufgeführt:

NUL	<i>null</i>	Nullzeichen, leeres Zeichen
SOH	<i>start of heading</i>	Anfang des Kopfes
STX	<i>start of text</i>	Anfang des Textes
ETX	<i>end of text</i>	Ende des Textes
EOT	<i>end of transmission</i>	Ende der Übertragung
ENQ	<i>enquiry</i>	Stationsaufforderung
ACK	<i>acknowledge</i>	Positive Rückmeldung
BEL	<i>bell</i>	Klingelzeichen
BS	<i>backspace</i>	Rückwärtsschritt
HT	<i>horizontal tabulation</i>	Horizontal-Tabulator
LF	<i>line feed</i>	Zeilenvorschub
VT	<i>vertical tabulation</i>	Vertikal-Tabulator
FF	<i>form feed</i>	Seitenvorschub
CR	<i>carriage return</i>	Wagenrücklauf
SO	<i>shift out</i>	Dauerumschaltung
SI	<i>shift in</i>	Rückschaltung
DLE	<i>data link escape</i>	Datenübertragungsumschaltung
DCi	<i>device control</i>	Gerätesteuerung
NAK	<i>negative acknowledge</i>	Negative Rückmeldung
SYN	<i>synchronous idle</i>	Synchronisierung
ETB	<i>end of transmission block</i>	Ende des Datenübertragungsblockes
CAN	<i>cancel</i>	Ungültig
EM	<i>end of medium</i>	Ende der Aufzeichnung
SUB	<i>substitute character</i>	Substitution
ESC	<i>escape</i>	Fluchtzeichen (Umschaltung)
FS	<i>file separator</i>	Hauptgruppen- bzw. Dateitrennung
GS	<i>group separator</i>	Gruppentrennung
RS	<i>record separator</i>	Untergruppen- bzw. Satztrennung
US	<i>unit separator</i>	Untergruppentrennung
SP	<i>space</i>	Leerzeichen
DEL	<i>delete</i>	Löschen eines Zeichens

Tabelle 5: ASCII-Steuerzeichen

3.3.1 Zeichenkonstanten

Zeichenkonstanten in C-Quelltexten sind in einfache Hochkommas eingeschlossene Zeichen, etwa 'K' für das Zeichen K. Eine Zeichenkonstante wird als ganzzahliger Wert aufgefasst — der Wert des Zeichens im Maschinensatz. (Die Zeichenkonstante 'K' hat somit den dezimalen Wert 75, s.o.)

Für einige Zeichen (insbes. Steuerzeichen) müssen in C-Quelltexten (ebenfalls in einfache Hochkommas einzuschließende) Ersatzdarstellungen verwendet werden:

<code>\n</code>	newline	<code>\t</code>	horizontal tab	<code>\v</code>	vertical tab	<code>\b</code>	backspace
<code>\r</code>	carriage return	<code>\f</code>	formfeed	<code>\a</code>	bell	<code>\\</code>	backslash
<code>\?</code>	question mark	<code>\'</code>	single quote	<code>\"</code>	double quote		

Auf diese Art und Weise können nur die Zeichen des Maschinensatzes als Zeichenkonstanten dargestellt werden (Werte liegen im Bereich von 0 bis 127!).

Beliebige Werte (von 0 bis 255 bzw. von -127 bis 127) können oktal in der Form `'\ooo'`, wobei *ooo* eine Folge von ein bis drei oktalen Ziffern ist, oder hexadezimal in der Form `'\xhh'`, wobei *hh* eine Folge von ein bis zwei hexadezimalen Ziffern ist, dargestellt werden. Folgende Zeichenkonstanten sind also gleichwertig: `'K'`, `'\113'` und `'\x4B'`.

3.3.2 Definition und Initialisierung von Variablen vom Typ `char`

Die Definition und Initialisierung von Variablen vom Typ `char` erfolgt wie bei Variablen von anderem Typ:

```
...
void main(void)
{
    char c, d='K';
    unsigned char e='\113', f='\xFF';
    signed char g;
    ...
}
```

3.3.3 Operationen mit `char`-Werten

Werte vom Typ `char`, `signed char` oder `unsigned char` werden in Programmen wie kleine ganze Zahlen behandelt und somit sind für sie die üblichen ganzzahligen Operationen erlaubt (auch %)!

Ist ein Wert vom Typ `char` (`signed` oder `unsigned`) ein Operand in einem Ausdruck, so wird dieser Wert zur Berechnung des Ausdruckes zumindest in den Typ `int` (ggf. noch "höher") umgewandelt (ganzzahlige Aufwertung wie bei `short`-Werten!).

Derartiges Rechnen mit `char`-Werten ist zwar nicht deren Hauptanwendung — hin- und wieder dennoch praktisch, etwa:

```
char c, d;
...
d = c + 'a' - 'A';
...
```

Die Variable `d` erhält als Wert den um `'a' - 'A'` erhöhten Wert der Variablen `c`. Die Zeichenkonstante `'a'` hat (im ASCII-Zeichensatz) den Wert 97, `'A'` den Wert 65 und somit hat `'a' - 'A'` den Wert $97 - 65 = 32$. Dieser Wert 32 wird auf den Wert

von `c` addiert. Nehmen wir an, dass `c` ein Großbuchstabe ist, etwa `K` (der dezimale Wert von `c` ist also 75), so hat `d` den dezimalen Wert $75 + 32 = 107$ — hierdurch wird im ASCII-Zeichensatz der Kleinbuchstabe `k` repräsentiert.

Da im ASCII-Zeichensatz alle Großbuchstaben unmittelbar hintereinander liegen (an den Positionen 65 bis 90) und alle Kleinbuchstaben auch unmittelbar hintereinander folgen (Positionen 97 bis 122), ist der “Abstand“ zwischen einem Großbuchstaben und dem korrespondierenden Kleinbuchstaben immer gleich und der entspricht dem “Abstand“ `'a' - 'A'`. Somit “berechnet“ der Ausdruck `c + 'a' - 'A'` zu einem Großbuchstaben den korrespondierenden Kleinbuchstaben! (Dies funktioniert mit jedem Maschinensatz, in dem die Großbuchstaben und die Kleinbuchstaben jeweils alphabetisch unmittelbar aufeinanderfolgend dargestellt werden — unabhängig von den tatsächlichen Positionen der Groß- und Kleinbuchstaben!)

3.3.4 Ein-/Ausgabe von Zeichen

Zur Eingabe von Zeichen haben wir bereits die Bibliotheksfunktion `int getchar(void)`; kennengelernt. Sie liest das nächste Zeichen vom Eingabefließband (Leerzeichen o.ä. werden hierbei nicht überlesen) und gibt als Funktionsergebnis den Wert des gelesenen Zeichens zurück. Das gelesene Zeichen wird hierbei als `unsigned char` interpretiert, der entsprechende (nicht negative) Wert wird aber als int-Wert im Funktionsergebnis zurückgeliefert!

Soll durch `getchar` der Wert einer `char`-Variablen `c` eingelesen werden, so braucht der Funktion der Name der Variablen nicht übergeben zu werden, sondern die Verwendung ist wie folgt:

```
c = getchar();
```

Die Funktion liest ein Zeichen vom Eingabefließband, gibt den Wert des gelesenen Zeichens als Funktionsergebnis (vom Typ `int`) zurück und dieser zurückgegebene Wert wird hier der `char`-Variablen `c` zugewiesen. (Hierbei findet implizit eine Typumwandlung statt: das Funktionsergebnis vom Typ `int` wird für die Zuweisung an die `char`-Variable in einen `char`-Wert umgewandelt! Liegt etwa ein `K` auf dem Eingabefließband an, so wird der `int`-Wert 75 zurückgeliefert und aus dieser `int`-75 wird zur Zuweisung eine `char`-75!)

Ist das Eingabefließband leer (noch nicht “geschlossen“), so hält das Programm an und wartet auf die nächste Eingabe.

Ist das Eingabefließband leer und geschlossen (es kann dann kein Zeichen mehr “kommen“), so gibt die Funktion `getchar` einen `int`-Wert zurück, der nicht der Wert eines Zeichens sein kann. (Dies ist der Grund dafür, dass `getchar` `int`-Werte und keine `char`-Werte liefert!) Für diesen als Zeichen unmögliche `int`-Wert (meistens `-1`) gibt es in C die vom System vordefinierte symbolische Konstante `EOF` (kann wie jede ganzzahlige Konstante verwendet werden!). (Gibt also die Funktion `getchar` den Wert `EOF` zurück, so ist das Eingabefließband leer und geschlossen!)

Aus diesem Grund ist das obige Einlesen eines Zeichens in eine `char`-Variable :

```
c = getchar();
```

gefährlich, da, wenn das Eingabefließband leer und geschlossen ist, von `getchar` der `int`-Wert `EOF` (`-1`) geliefert wird, bei der Zuweisung an `c` diese `-1` (vom Typ `int`) jedoch in einen `char`-Wert, also ggf. in einen positiven Wert, umgewandelt wird. Der in `c` vorliegende Wert würde dann das ordnungsgemäße Einlesen (irgend-)eines Zeichens vorgaukeln, ohne dass das Ende der Eingabe zur Kenntnis genommen wurde. Es ist in jedem Fall besser, den Ergebniswert von `getchar` einer `int`-Variablen zuzuweisen und dann mit den im nächsten Kapitel 4 vorgestellten Möglichkeiten zu untersuchen, ob tatsächlich ein Zeichen gelesen oder das Ende der Eingabe erreicht wurde.

Zur Eingabe von Zeichenwerten steht auch die `scanf`-Funktion mit der Eingabespezifikation `%c` zur Verfügung. Hierbei muss wieder der Name der einzulesenden Variablen (mit voranstehendem `&`) als Argument übergeben werden: `scanf("%c", &c);`.

Auch hierbei werden wie bei `getchar` führende Leerzeichen o.ä. auf dem Eingabefließband nicht überlesen. Ob das Lesen geklappt hat oder nicht kann man wie immer am Funktionsergebnis von `scanf` (Anzahl der Zuweisungen) ablesen!

Zur Ausgabe von Zeichenwerten kann `printf` mit der Ausgabespezifikation `%c` oder aber auch die zu `getchar` analoge Funktion:

```
int putchar(int);
```

verwendet werden. An dieser *Deklaration* (Bekanntmachung) der Funktion `putchar` kann man ablesen, dass beim Aufruf dieser Funktion ein Argument vom Typ `int` in den runden Klammern anzugeben ist und dass diese Funktion wiederum ein Funktionsergebnis vom Typ `int` liefert. (Anhand dieser Deklaration kann der Compiler feststellen, ob die Funktion korrekt aufgerufen wurde. Die Deklarationen der Ein-/Ausgabefunktionen der Standardbibliothek stehen in `stdio.h`!)

Das zum Wert des übergebenen Argumentes (Typ `int`) gehörende Zeichen wird auf dem Bildschirm ausgegeben und (zur Kontrolle) wird dieser Wert (oder `EOF`, falls bei der Ausgabe ein Fehler aufgetreten ist!) nochmals als Funktionsergebnis zurückgeliefert!

Beispiel: `i = putchar('K');`

`'K'` hat den Wert 75 (Typ `char`), dieser Wert wird in `int` umgewandelt (der Compiler weiß ja, dass `putchar` ein Argument vom Typ `int` benötigt!) und das zum Wert 75 gehörende Zeichen (also `K`) wird ausgegeben. Die Variable `i` hat anschließend, falls kein Fehler aufgetreten ist, ebenfalls den Wert 75 haben (sonst `EOF`)!

4 Ablaufkontrolle

Die in einem C-Programm stehenden Anweisungen werden i. Allg. alle hintereinander ausgeführt.

Bisweilen ist es wünschenswert, einige Anweisungen mehrfach oder andere nur in gewissen Fällen auszuführen.

In allen höheren Programmiersprachen gibt es hierfür Sprachmittel, Kontrollstrukturen genannt!

4.1 Auswahlanweisungen

4.1.1 Die einfache if-Anweisung

Die einfache if-Anweisung hat die Form:

```
if (Bedingung)
    Anweisung
```

Die *Anweisung* (hier kann eine beliebige Anweisung stehen) wird nur dann ausgeführt, wenn die *Bedingung wahr* ist, etwa:

Eine Division soll etwa nur dann durchgeführt werden, wenn die Zahl, durch die geteilt werden soll, ungleich 0 ist, oder

wird eine Zahl mit `scanf` eingelesen, so soll erst dann mit ihr weitergerechnet werden, wenn das Einlesen tatsächlich geklappt hat.

Eine Bedingung in C ist ein beliebiger Ausdruck, der einen Zahlwert liefert. Ist der Zahlwert ungleich dem Wert 0, so wird die Bedingung als wahr interpretiert. Ist der Zahlwert gleich 0, so wird die Bedingung als falsch interpretiert.

Im Kontext eines Programms könnte eine if-Anweisung wie folgt aussehen:

```
Anweisung1
if (Bedingung)
    Anweisung2
Anweisung3      } Auswahlanweisung
```

Kommt der Programmfluss an *Anweisung1* an, so wird diese ausgeführt. Anschließend wird die *Bedingung* der Auswahlanweisung ausgewertet — nur falls sie als *wahr* interpretiert wird (also ein Ergebnis ungleich 0 liefert), wird *Anweisung2* ausgeführt (diese heißt: “von der *Bedingung* abhängig oder kontrolliert”) — anschließend wird in jedem Fall *Anweisung3* ausgeführt.

Zur Formulierung von Bedingungen werden häufig Vergleichsoperatoren verwendet:

Operator	Bedeutung	Operator	Bedeutung
<	Test auf kleiner	<=	Test auf kleiner oder gleich
>	Test auf größer	>=	Test auf größer oder gleich
==	Test auf Gleichheit	!=	Test auf Ungleichheit

Diese Vergleichsoperatoren liefern als Ergebnis eine 1, wenn der Vergleich zutrifft, und sonst eine 0. Der Ausdruck `i <= 10` hat beispielsweise genau dann den Wert 1, wird also genau dann als *wahr* interpretiert, wenn der Wert der Variablen `i` kleiner oder gleich 10 ist. Ansonsten hat er den Wert 0, wird also als *falsch* interpretiert!

Weiterhin stehen die *logischen Operatoren* zur Verfügung:

Operator	Bedeutung
<code>&&</code>	logische <i>Und</i> -Verknüpfung
<code> </code>	logische <i>Oder</i> -Verknüpfung
<code>!</code>	Negation

Der (zusammengesetzte) Ausdruck `ausdr1 && ausdr2` liefert als Ergebnis genau dann den Wert 1 (wird also als *wahr* interpretiert), wenn sowohl `ausdr1` als auch `ausdr2` als *wahr* interpretiert werden (d.h. wenn sowohl `ausdr1` als auch `ausdr2` jeweils einen Wert ungleich 0 haben)! Ansonsten hat `ausdr1 && ausdr2` den Wert 0!

Hier tritt eine Besonderheit ein: in `ausdr1 && ausdr2` wird zunächst `ausdr1` ausgewertet, um festzustellen, ob dieser *wahr* ist. Ist `ausdr1` jedoch schon *falsch*, so ist der Gesamtausdruck `ausdr1 && ausdr2` auf jeden Fall *falsch*, unabhängig davon, ob `ausdr2` nun *wahr* ist oder nicht! Auf die Auswertung von `ausdr2` kann verzichtet werden (und es wird!).

Der (zusammengesetzte) Ausdruck `ausdr1 || ausdr2` liefert als Ergebnis genau dann den Wert 1 (wird also als *wahr* interpretiert), wenn `ausdr1` oder `ausdr2` (oder auch beide) als *wahr* interpretiert werden (d.h. wenn `ausdr1` oder `ausdr2` oder auch beide einen Wert ungleich 0 haben)! Ansonsten hat `ausdr1 || ausdr2` den Wert 0!

Hier tritt die Besonderheit ein: in `ausdr1 || ausdr2` wird zunächst `ausdr1` ausgewertet, um festzustellen, ob dieser *wahr* ist. Ist `ausdr1` jedoch schon *wahr*, so ist der Gesamtausdruck `ausdr1 || ausdr2` auf jeden Fall *wahr*, unabhängig davon, ob `ausdr2` nun *wahr* ist oder nicht! Auf die Auswertung von `ausdr2` kann verzichtet werden (und es wird!).

Der Negationsoperator `!` hat nur einen Operanden: `!ausdr`.

Hat `ausdr` einen Wert ungleich 0 (ist also als *wahr* zu interpretieren), so liefert `!ausdr` den Wert 0 (wird also als *falsch* interpretiert)! Hat `ausdr` den Wert 0 (ist also als *falsch* zu interpretieren), so liefert `!ausdr` den Wert 1 (wird also als *wahr* interpretiert)!

Es können komplexere Bedingungen formuliert werden, wobei auf die Priorität der Operatoren geachtet werden muss! Ggf. muss man durch (runde) Klammerung die Reihenfolge der Auswertung steuern!

Die bei `&&` bzw. `||` angemerkte Besonderheit hat u.U. Einfluss auf den weiteren Programmablauf, etwa in folgender Bedingung:

```
i < 10 && getchar() != EOF
```

Ist hierbei `i` tatsächlich kleiner als 10, also der Ausdruck links von `&&` *wahr*, so wird der Ausdruck `getchar() != EOF` rechts von `&&` "ausgewertet", d.h. es wird durch die Funktion `getchar()` ein Zeichen vom Eingabefließband gelesen und entfernt und das Funktionsergebnis (der Wert des gelesenen Zeichens) wird mit `EOF` verglichen.

Ist jedoch `i` nicht kleiner als 10, so ist der Gesamtausdruck `i < 10 && getchar() != EOF`

auf jeden Fall *falsch* und der Ausdruck `getchar() != EOF` wird nicht ausgewertet, d.h. die Funktion `getchar()` wird auch nicht aufgerufen!

Je nachdem, ob der linke Ausdruck `i < 10` wahr ist oder nicht, wird also die Funktion `getchar()` aufgerufen oder nicht. Dies hat u.U. Konsequenzen (unterschiedliche Seiteneffekte) für weitere Leseoperationen!

Häufig ist es der Fall, dass, falls die *Bedingung* als *wahr* interpretiert wird, nicht nur eine, sondern gleich mehrere Anweisungen durchgeführt werden sollen, und falls sie *falsch* ist, keine von diesen! (D.h. es sollen mehrere Anweisungen von der *Bedingung* kontrolliert werden!)

Hierzu muss man die Anweisungen, welche gemeinsam von der *Bedingung* abhängen sollen, durch geschweifte Klammern zu einer sog. Verbundanweisung (engl: *Compound-Statement*) zusammenfassen:

```

if (Bedingung)
{
    Anweisung1
    Anweisung2
    :
}

```

Verbundanweisung

Eine Verbundanweisung zählt als eine Anweisung. Zur Erhöhung der Lesbarkeit des Quelltextes positioniert man gewöhnlich die zugehörigen geschweiften Klammern so, dass sie auffallen, und rückt die durch die Klammern zusammengefassten Einzelanweisungen konsistent ein! (Strenggenommen ist der Anweisungsteil des Hauptprogramms — also alles von der geschweiften Klammer auf hinter der Kopfzeile mit `main` bis zur schließenden geschweiften Klammer hinter `return;` — auch eine Verbundanweisung!) Ein Beispiel für die einfache `if`-Anweisung ist in `ablauf/if_else/if_einf.c`.

4.1.2 Die `if-else`-Anweisung

Die Funktionsweise der `if-else`-Anweisung:

```

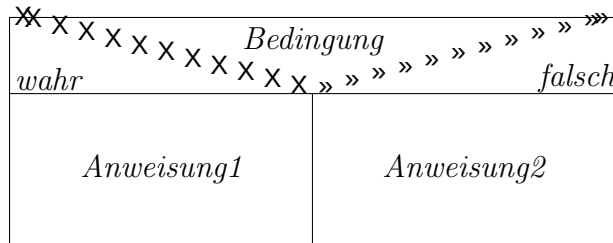
if (Bedingung)
    Anweisung1
else
    Anweisung2

```

ist wie folgt: Ist die Bedingung *wahr*, so wird *Anweisung1* ausgeführt und *Anweisung2* nicht — ist die Bedingung jedoch *falsch*, so wird *Anweisung1* nicht ausgeführt, dafür jedoch *Anweisung2*. (Genau eine dieser beiden Anweisungen wird ausgeführt — welche wird anhand der Bedingung entschieden!)

Anweisung1 und *Anweisung2* können (unabhängig voneinander) einfache Anweisungen oder Verbundanweisungen sein! *Anweisung1* heißt der `if`-Teil und *Anweisung2* der `else`-Teil der `if-else`-Anweisung.

Für derartige `if-else`-Anweisungen gibt ein Struktogrammelement:

Abbildung 17: Struktogrammelement: `if-else`-Anweisung

(Für die einfache `if`-Anweisung gibt es kein eigenes Struktogrammelement, man behilft sich dann mit dem für eine `if-else`-Anweisung, indem man für den `else`-Teil (*Anweisung2*) die sog. leere Anweisung `%` einträgt!)

Ein Beispiel für eine `if-else`-Anweisung ist in `ablauf/if_else/if_else.c`.

4.1.3 Geschachtelte `if`-Anweisungen

Die in einer `if`-Anweisung von der Bedingung kontrollierte Anweisung (`if`-Teil und ggf. `else`-Teil) kann — wie bereits erwähnt — eine beliebige Anweisung sein: eine einfache Anweisung, eine Verbundanweisung oder auch eine `if`- bzw. `if-else`-Anweisung.

Vorsicht ist geboten, wenn der `if`-Teil einer `if-else`-Anweisung eine einfache `if`-Anweisung sein soll! Hier gilt die Regel, dass sich ein `else` immer auf das unmittelbar davor stehende `if` bezieht:

In

```

if (Bedingung_1)
    if (Bedingung_2)
        Anweisung_1
    else
        Anweisung_2

```

bezieht sich das `else` auf das zweite `if` (mit *Bedingung_2*) und nicht, wie vom Programmierer offensichtlich gewünscht (durch Einrückung deutlich gemacht!) auf das erste `if` (mit *Bedingung_1*). D.h. hier gehört `if (Bedingung_1)` zu einer einfachen `if`-Anweisung, deren kontrollierte Anweisung (`if`-Teil) eine `if-else`-Anweisung ist. Soll (wie hier augenscheinlich vom Programmierer gewünscht!) `if (Bedingung_1)` zu einer `if-else`-Anweisung gehören, deren `if`-Teil eine einfache `if`-Anweisung (mit von dieser kontrollierten *Anweisung_1*) und deren `else`-Teil *Anweisung_2* ist, so muss man das durch geschweifte Klammern deutlich machen:

```

if (Bedingung_1)
{
    if (Bedingung_2)
        Anweisung_1
    }
else

```


Zählschleifen motivierende (hoffentlich!) und demonstrierende Beispiele sind die Quelltexte `summe1.c` bis `summe4.c` im Verzeichnis `ablauf/schleife/for`.

Die Zählschleife, auch for-Schleife genannt, hat folgende Form:

```
for ( Initialisierung; Bedingung; Inkrementierung )
    Anweisung
```

und in der Anwendung sieht sie beispielsweise so aus:

```
...
int i;
...
for ( i = 0; i < 100; i = i + 1 )
    printf("Ich soll meine Hausaufgaben machen.\n");
...
```

Hierdurch wird (“zur Erledigung einer Strafarbeit“) hundert mal der Satz

`Ich soll meine Hausaufgaben machen.`

ausgegeben. Etwas genauer:

Die `int`-Variable `i` bekommt zunächst den Wert 0 (*Initialisierung* `i = 0`, wird genau einmal vor Beginn der eigentlichen Schleife durchgeführt), anschließend wird überprüft, ob `i` kleiner als 100 ist (die *Bedingung* `i < 100` wird überprüft!). Trifft diese zu (d.h. die *Bedingung* ist *wahr*) so wird die *Anweisung* (hier der `printf`-Befehl) ausgeführt und anschließend wird `i` um 1 erhöht (`i = i + 1`, die *Inkrementierung* wird durchgeführt). Hiernach wird erneut die Bedingung (`i < 100`) überprüft, ist sie noch immer *wahr*, so wird erneut die Anweisung und anschließend die Inkrementierung durchgeführt usw..

Die Schleife bricht ab, wenn zum erstenmal die *Bedingung* nicht mehr zutrifft, also *falsch* wird — dann geht es sofort (ohne weitere *Inkrementierung*) mit der nächsten Anweisung hinter der Zählschleife weiter!

Initialisierung, *Bedingung* und *Inkrementierung* sind beliebige Ausdrücke (Verknüpfung von Operanden — also Variablen, Konstanten oder anderen Ausdrücken — durch Operatoren) und diese steuern den Ablauf der Schleife: wie es anfängt (*Initialisierung*), wie lange es geht (*Bedingung*) und mit welcher “Schrittweite“ (*Inkrementierung*) es vorangeht! Sowohl die *Initialisierung* als auch die *Bedingung* als auch die *Inkrementierung* dürfen fehlen (wenn es Sinn hat! Vorsicht: man hat schnell eine Endlosschleife programmiert!). Die Strichpunkte dürfen nicht fehlen. Eine fehlende Bedingung wird immer als *wahr* interpretiert!

Die Schleife kontrolliert wieder genau eine (aber beliebige) Anweisung. Die kontrollierte Anweisung kann natürlich auch eine `if`-Anweisung und auch selbst wieder eine Schleife sein!

Möchte man, dass mehrere Anweisungen in Abhängigkeit von der Schleife ausgeführt werden sollen, so muss man diese wieder durch geschweifte Klammern zu einer Verbundanweisung zusammenfassen.

Bisweilen ist der Anweisungsteil einer Zählschleife auch leer (siehe etwa das Beispiel auf Seite 91), wenn durch Initialisierung, Bedingung und Inkeremtierung bereits alles

erforderliche erledigt. Die leere Anweisung besteht nur aus einem Strichpunkt, ohne einen davorstehenden Ausdruck!.

Beispiele für einfache Zählschleifen:

- Summe der Quadrate aller Zahlen von 1 bis 100 berechnen:

```
int i, n = 100;
long summe; /* koennte ziemlich gross werden! */
...
summe = 0;
for ( i = 1; i <= n; i = i + 1)
    summe = summe + i*i;
...
```

- Summe der Quadrate aller geraden Zahlen von 1 bis 100 berechnen:

```
int i, n = 100;
long summe; /* koennte ziemlich gross werden! */
...
summe = 0;
for ( i = 2; i <= n; i = i + 2)
    summe = summe + i*i;
...
```

- Alle Großbuchstaben in umgekehrter Reihenfolge ausgeben:

```
char c;
...
for (c = 'Z'; c >= 'A'; c = c - 1)
    putchar(c);
...
```

Zur Darstellung von derartigen Schleifen in Struktogrammen gibt es folgendes Struktogrammelement:

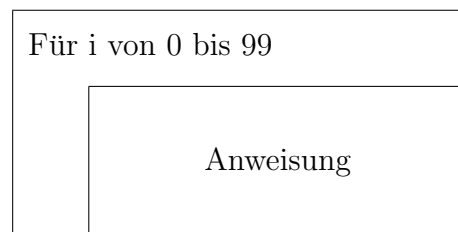


Abbildung 19: Struktogrammelement: Zählschleife

Im Anweisungsteil einer Zählschleife (üblicherweise dann eine Verbundanweisung) können zur Handhabung von Sonderfällen die Anweisungen `continue` bzw. `break` stehen. Ihre Anwendung ist i. Allg. wie folgt:

- `if ( sonderfall ) continue;`

Tritt der Sonderfall ein, so wird die Anweisung `continue`; ausgeführt. Hierdurch wird der Anweisungsteil der Schleife sofort beendet (verlassen), es wird sofort die *Inkrementierung* durchgeführt und anschließend die *Bedingung* der Schleife

fe erneut überprüft (ggf. geht es dann mit einem weiteren Schleifendurchlauf weiter!).

- `if ( sonderfall ) break;`

Tritt der Sonderfall ein, so wird die Anweisung `break;` ausgeführt. Hierdurch wird der Anweisungsteil der Schleife sofort beendet (verlassen) und es geht mit der nächsten Anweisung hinter der Schleife weiter (keine *Inkrementierung* und keine erneute Überprüfung der *Bedingung*!).

Ein `break` bzw. ein `continue` bezieht sich bei geschachtelten Schleifen immer auf die innerste Schleife, in der sie stehen!

`break` und `continue` sind im Beispiel `ablauf/schleife/for/summe5.c` demonstriert.

4.3 Die `while`-Schleife

Die `while`-Schleife hat folgende Form:

```
while (Bedingung)
    Anweisung
```

Die *Bedingung* wird überprüft, trifft diese zu, so wird die *Anweisung* durchgeführt, anschließend die *Bedingung* erneut überprüft und ggf. die *Anweisung* erneut ausgeführt usw. bis zum erstenmal die *Bedingung* nicht mehr zutrifft.

Die Anweisung kann eine beliebige Anweisung sein.

Das zugehörige Struktogrammelement ist das gleiche wie für `for`-Schleifen (diese beiden Schleifenarten sind nahezu gleichartig zu verwenden!):

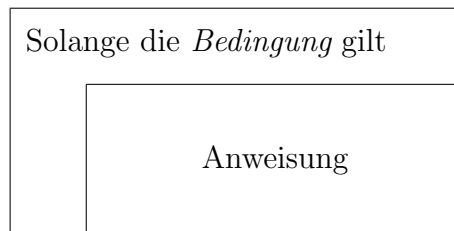


Abbildung 20: Struktogrammelement: kopfgesteuerte Schleife

Auch für den Anweisungsteil einer `while`-Schleifen gibt es die Sonderanweisungen `break` und `continue` mit der naheliegenden Bedeutung:

- `if ( sonderfall ) continue;`

Tritt der Sonderfall ein, so wird die Anweisung `continue;` ausgeführt. Hierdurch wird der Anweisungsteil der Schleife sofort beendet (verlassen) und anschließend sofort die *Bedingung* der Schleife erneut überprüft (ggf. geht es dann mit einem weiteren Schleifendurchlauf weiter!).

- `if ( sonderfall ) break;`

Tritt der Sonderfall ein, so wird die Anweisung `break;` ausgeführt. Hierdurch wird der Anweisungsteil der Schleife sofort beendet (verlassen) und es geht mit der nächsten Anweisung hinter der Schleife weiter (keine erneute Überprüfung der *Bedingung*!).

Beispiel `ablauf/schleife/while/while.c` demonstriert die Verwendung einer `while`-Schleife.

4.4 Die `do-while`-Schleife

Sowohl bei der `for`-Schleife als auch bei der `while`-Schleife wird, bevor der Anweisungsteil durchgeführt wird, zunächst die Bedingung überprüft. Ggf. wird der Anweisungsteil dann keinmal ausgeführt. Derartige Schleifen heißen abweisende Schleifen oder kopfgesteuerte Schleifen (Bedingung steht “oben“!).

Im Gegensatz dazu ist die `do-while`-Schleife eine fußgesteuerte Schleife, d.h. die Bedingung steht “unten“:

```
do
    Anweisung
while (Bedingung);
```

(Achtung: Strichpunkt hinter der schließenden runden Klammer zur *Bedingung* nicht vergessen!)

Hier wird die *Anweisung* zunächst einmal durchgeführt und anschließend wird die *Bedingung* überprüft. Trifft diese zu, so wird die *Anweisung* erneut ausgeführt und anschließend wird erneut die *Bedingung* überprüft usw. bis zum erstenmal die *Bedingung* nicht mehr zutrifft!

I. Allg. wird der Anweisungsteil (kann wiederum eine beliebige Anweisung, ggf. Verbundanweisung sein!) also mindestens einmal ausgeführt.

Das zugehörige Struktogrammelement ist:

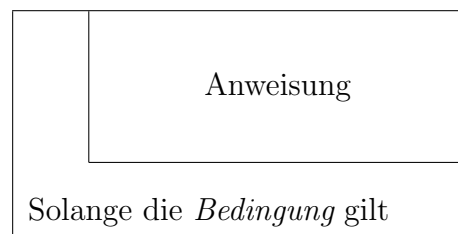


Abbildung 21: Struktogrammelement: fußgesteuerte Schleife

Auch hier gibt es wieder die beiden Sonderanweisungen `break` und `continue` mit der gleichen Bedeutung wie bei der `while`-Schleife.

Die `do-while`-Schleife wird in den Programmen `ablauf/schleife/while/do_while.c` und in `ablauf/beispiel/leibniz.c` demonstriert!

4.5 Die `switch`-Anweisung

Mit der `switch`-Anweisung kann man, ähnlich zu einer geschachtelten `if-else`-Anweisung, eine Fallunterscheidung zwischen mehreren Fällen behandeln.

Dabei ist die generelle Form dieser Anweisung noch ein wenig allgemeiner, so dass

erst eine spezielle Form diese klassische Fallunterscheidung realisiert. Die allgemeine Form ist:

<pre>switch (<i>Ausdruck</i>) { case <i>k_ausd_1</i>: _____ _____ case <i>k_ausd_2</i>: _____ _____ case <i>k_ausd_3</i>: _____ case <i>k_ausd_4</i>: _____ _____ : _____ case <i>k_ausd_n</i>: _____ _____ _____ default: _____ _____ }</pre>	Eine Folge von beliebigen Anweisungen. Einige, nicht unbedingt jede dieser Anweisungen sind mit sog. <u>case-Labeln</u> versehen, d.h. unmittelbar vor der Anweisung steht case <i>Konstanter_Ausdruck</i>: (an diesen Ausdrücken dürfen keine Variablen beteiligt sein!) und maximal eine der Anweisungen ist mit dem Label default: versehen. (Falls dieser Label auftritt, ist er gewöhnlich der letzte auftretende Label!) Eine Anweisung kann auch mit mehreren Labeln versehen sein!
--	---

Die Funktionsweise der **switch**-Anweisung ist wie folgt:

Der *Ausdruck* in den runden Klammern hinter dem **switch** wird ausgewertet und der Wert des Ausdrucks wird der Reihe nach mit den (konstanten) Ausdrücken der auftretenden Label verglichen, bis zum erstenmal eine Übereinstimmung vorliegt. Bei dieser ersten Übereinstimmung mit einem Label-Ausdruck werden die zu diesem Label gehörende Anweisung und alle darunterstehenden Anweisungen durchgeführt.

Mit dem **default**-Label "stimmt" jeder Wert "überein"! Sollte also innerhalb der **switch**-Verbundanweisung ein **default**-Label vorkommen, so wird spätestens bei diesem "Übereinstimmung festgestellt" und die zugehörige Anweisung und alle folgenden werden in jedem Fall ausgeführt!

(Man kann sich eine derartige **switch**-Anweisung anschaulich als einen "langen Flur von Anweisungen" vorstellen und die Label sind "die Türen zu diesem Flur". Anhand des Ausdrucks wird entschieden, durch welche Tür das Programm "in den Flur hineinkommt" und somit welche der Anweisungen es bis "zum Ende des Flurs" noch ausführen muss!)

Solche **switch**-Anweisungen entsprechen nicht ganz den Prinzipien der Strukturierten Programmierung und es gibt kein entsprechendes Struktogrammelement.

Trotzdem gibt es nützliche Anwendungen, siehe etwa das Programm `ablauf/switch/switch1.c`.

(In diesem Beispiel kommt eine besondere Form einer Formatzeichenkette beim Aufruf von **scanf** vor:

```
scanf("%d.%d.%d",&tag, &monat, &jahr);
```

— hierbei sind *tag*, *monat* und *jahr* jeweils Variablen vom Typ **int**, die hiermit

eingelezen werden sollen! Das besondere an der Formatzeichenkette "%d.%d.%d" sind die Punkte. Diese gehören nicht zu den Eingabespezifikationen %d, sondern treten als "gewöhnliche" Zeichen auf.

Tritt in einer Formatzeichenkette ein gewöhnliches Zeichen auf, so wird erwartet, dass auch in der Eingabe an entsprechender Stelle genau dieses Zeichen steht und dieses Zeichen wird dann gelesen und vom Eingabefließband entfernt! Fehlt das Zeichen in der Eingabe, so kann der Lesevorgang nicht korrekt durchgeführt werden! Der obige Einlesebefehl liest somit drei ganzzahlige Werte, wobei jeweils unmittelbar hinter den ersten beiden ganzzahligen Werten auf der Eingabe ein Punkt stehen muss!)

Um mit der **switch**-Anweisung reine Fallunterscheidungen zu programmieren, muss man innerhalb der Liste von Anweisungen die **break**-Anweisung benutzen! Tritt im Laufe der Abarbeitung der Anweisungsliste einer **switch**-Anweisung die Anweisung **break**; auf, so wird die gesamte **switch**-Anweisung sofort ganz verlassen. Eine so realisierte (reine) Fallunterscheidung sieht dann so aus:

```
switch (Ausdruck) {
    case Fall_1: ...
                break;
    case Fall_2: ...
                break;
    case Fall_3: ...
                break;
        :
        :
    case Fall_n: ...
                break;
    default: ...
            break;
}
```

Als Struktogrammelement wird das gleiche, wie bei geschachtelten **if-else**-Anweisungen verwendet (vgl. Seite 77):

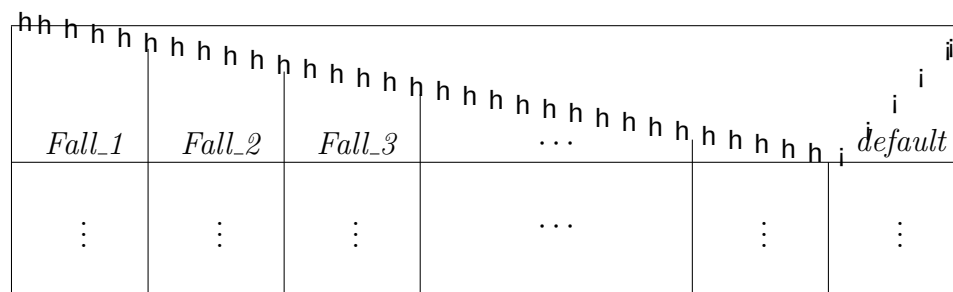


Abbildung 22: Struktogrammelement: **switch**-Anweisung

Der **default**-Label braucht auch hier nicht vorhanden zu sein!

In Programm `ablauf/switch/switch2.c` ist diese Art der **switch**-Anweisung demonstriert.

4.6 Die `goto`-Anweisung

Als letzte (und aus FORTRAN-Zeiten als Relikt übriggebliebene) Möglichkeit, den Ablauf eines Programms zu beeinflussen, bietet die `goto`-Anweisung.

Ähnlich wie bei den `case`-Labeln in der `switch`-Anweisung kann jede Anweisung in einem Programm mit einem Label versehen werden. Label sind Namen (Bezeichner) gefolgt von einem Doppelpunkt und stehen unmittelbar vor der zu “labelnden” Anweisung. (Namen von Labeln dürfen mit Namen von Variablen oder Funktionen, nicht jedoch mit Schlüsselwörtern übereinstimmen!)

Der Name eines derartigen Labels kann als Ziel einer `goto`-Anweisung angegeben werden. Bei Ausführung dieser `goto`-Anweisung wird der Programmablauf an dieser Stelle unterbrochen und es wird zu derjenigen Anweisung “gesprungen“, welche mit dem entsprechenden Label versehen ist und der Programmfluss wird mit dieser Anweisung und den ihr folgenden fortgesetzt.

Beispiel:

```
...
if ( sonderfall ) goto label_name;
.
.
.
labelname: gelabelte_anweisung;
...
```

Derartige `goto`-Anweisungen entsprechen nicht den Prinzipien der Strukturierten Programmierung und man kann immer so programmieren, dass man ohne sie auskommt.

In wenigen Sonderfällen werden sie in C-Programmen geduldet, etwa zum Verlassen einer geschachtelten Schleife in einer völlig verkorksten Situation. (Mit `break` würde man nur aus der innersten Schleife “herausspringen“ können!)

Ein (so nicht lauffähiges) Beispielfragment ist `ablauf/goto/goto.c`.

4.7 Weitere Beispiele zu Kontrollstrukturen

Weitere Beispiele sind als Quelltexte im Verzeichnis `ablauf/beispiele` abgelegt.

Eine Besonderheit wird in einigen dieser Programme (und in C-Programmen überhaupt gerne) verwendet:

Das Gleichheitszeichen `=` für die Zuweisung ist — wie bereits häufiger erwähnt — ein Operator, d.h. mit diesem können Ausdrücke formuliert werden!

Dies ist bereits schon häufiger ausgenutzt worden, etwa als Initialisierung und Inkrementierung in einer `for`-Schleife:

```
for ( i = 0; i < 100; i = i + 1 ) ...
```

Als Initialisierung und Inkrementierung müssen Ausdrücke stehen und `i = 0` bzw. `i = i + 1` sind Ausdrücke! (Derartige Zuweisungsausdrücke werden im Programm erst durch einen anschließenden Strichpunkt zu einer Anweisung!)

Zuweisungsausdrücke haben einen Wert und zwar den Wert der linken Seite nach der Zuweisung und dieser Wert kann weiterverwendet werden.

Genau wie der Ausdruck `i + 5` den Wert 15 hat, falls `i` den Wert 10 hat, und dieser Wert etwa in der Zuweisung `j = i + 5` weiterverwendet werden kann, hat der Ausdruck `i = i + 1` als Wert den erhöhten Wert von `i` und dieser Wert kann etwa in `printf("%d\n", i = i + 1);` weiterverwendet werden: `i` wird um 1 erhöht und der erhöhte Wert von `i` wird ausgegeben!

Oder im Ausdruck `i = j = 0;` (aufgrund der Assoziativität des Zuweisungsoperators von rechts nach links wird dies wie `i = ( j = 0 );` ausgewertet!) bekommt die Variable `j` den Wert 0 zugewiesen und die Variable `i` erhält als Wert den Wert der Zuweisung `j = 0`, also auch den Wert 0.

Dies wird in einigen der Beispielprogramme in der Bedingung einer `while`-Schleife verwendet:

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

Die Bedingung ist `(c = getchar()) != EOF`.

Es wird ein Zeichen gelesen (Aufruf von `getchar()`) und der Wert des gelesenen Zeichens wird der `int`-Variablen `c` zugewiesen (`c = getchar()`).

Anschließend wird der Wert dieser Zuweisung (also der Wert des gelesenen Zeichens) daraufhin überprüft, ob er mit `EOF` übereinstimmt oder nicht und abhängig hiervon ist der Ablauf der Schleife. (Man beachte, dass die Klammern um `c = getchar()` notwendig sind, da der Vergleichsoperator `!=` eine höhere Priorität hat als der Zuweisungsoperator `=` (vgl. Operatortabelle auf Seite 46 bzw. 138) und bei fehlenden Klammern der Ausdruck `c = getchar() != EOF` implizit wie `c = (getchar() != EOF)` ausgewertet würde. Dies ist grammatisch völlig korrekt, hat aber einen ganz anderen Sinn!)

Die (übersetzten) Programme `kopieren.c`, `zeichen.c`, `zeilen.c` und `worte.c` lesen jeweils von der Tastatur und schreiben auf den Bildschirm.

Wie in Abschnitt 1.2.6 bereits erwähnt, erlaubt es das UNIX-Betriebssystem durch geeignete Aufrufe eines Programms die Eingabe “von einer Datei umzulenken” und unabhängig davon die Ausgabe “auf eine (andere) Datei umzulenken”.

- Umlenken der Eingabe: `programmname < eingabedatei<cr>`

Das Programm läuft ab, dabei wird nicht von der Tastatur gelesen, sondern von der angegebenen Datei. In der Datei sollte dann genau dass stehen, was man auch interaktiv auf der Tastatur eingegeben hätte. Auf diese Weise kann man für ein Programm (unterschiedliche) “Eingaben” vorbereiten. Wird beim Lesen das Ende der Datei erreicht, so wird das Eingabefließband “geschlossen”.

- Umlenken der Ausgabe: `programmname > ausgabedatei<cr>`

Das Programm läuft ab, wobei die Ausgabe nicht auf dem Bildschirm erscheint, sondern in die angegebene, neue Datei geschrieben wird. Der vorherige Inhalt der Datei geht verloren.

Alternativ kann durch: `programmname >> ausgabedatei<cr>`

erreicht werden, dass die Ausgabe des Programms an die (eventuell bereits vor-

handene) angegebene Datei angehängt wird.

Ein- und Ausgabe können gleichzeitig umgelenkt werden!

Die C-Standardbibliothek stellt eigene Funktionen zur Behandlung von (gleichzeitig mehreren) Eingabe- und Ausgabedateien zur Verfügung. Hierauf wird in Abschnitt 11.1 näher eingegangen!

5 Felder

5.1 Grundlagen zu Feldern

Es tritt häufig auf, dass in Programmen sehr viele gleichartige Variablen (gleicher Typ) benötigt werden, mit denen im Programm gleichartige Dinge angestellt werden (gleiche Verwendung).

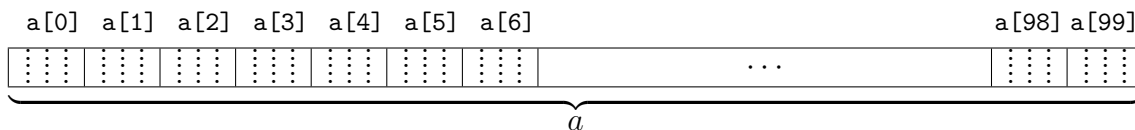
Es ist ziemlich mühsam (viel Schreibarbeit), etwa 100 `int`-Variablen zu definieren (100 Bezeichner ausdenken!) und diese 100 Variablen gleich zu verwenden!

In allen höheren Programmiersprachen gibt es die Möglichkeit, sog. *Felder* (auch *Vektoren* oder *Arrays* genannt) zu definieren, in denen mehrere gleichartige Variablen zu einem Objekt zusammengefasst werden. In C geschieht dies beispielsweise bei einer Variablendefinition durch:

```
int a[100];
```

Das Objekt `a` ist ein *Feld der Länge 100 vom Typ `int`*, durch diese Definition sind auf einen Schlag 100 `int`-Variablen definiert. Diese haben die Namen: `a[0]`, `a[1]`, ..., `a[99]`.

Durch die Definition `int a[100];` wird also Speicherplatz reserviert, der ausreicht, um 100 ganzzahlige Werte (Typ `int`) abzuspeichern. Bei 4 Byte pro `int` sind somit 400 Bytes reserviert:



Der Zugriff auf die einzelnen Variablen erfolgt durch Angabe des Feldnamens und sofort dahinter in eckigen Klammern des *Feldindexes*. Der Bereich der zulässigen *Feldindizes* ist hier von 0 bis 99, dieser Bereich fängt immer mit 0 an und endet 1 vor der bei der Felddefinition angegebenen Feldlänge!

Der Index selber darf hierbei ein beliebiger ganzzahliger Ausdruck (Typ `char`, `short`, `int` oder `long`, jeweils `signed` oder `unsigned`) sein, etwa `a[ 25 - 12 ]` oder `a[i]`, falls `i` etwa eine `int`-Variable ist!

Der Programmierer muss selbst dafür sorgen, dass seine Indexausdrücke in dem für das geg. Feld zulässigen Indexbereich liegen, d.h. etwa für unser Feld `a`, dass sie zwischen 0 und 99 liegen!

Liegt ein Index nicht im zulässigen Bereich, so spricht man von einem *Feldüberlauf*. Diese werden vom Compiler nicht entdeckt und können zur Laufzeit merkwürdigste Fehler nach sich ziehen!

Im Programm können die einzelnen Feldelemente wie gewöhnliche Variablen vom entsprechenden Typ verwendet werden. Man kann nicht auf ein Feld als ganzes zugreifen, sondern nur auf die einzelnen Feldelemente. Insbesondere ist die Zuweisung ganzer Felder nicht möglich:

```
int a[100], b[100];
...
a = b; /* Fehler!!!!*/
```

Zur Bearbeitung aller Elemente eines Feldes bieten sich Zählschleifen an. In folgenden Beispielfragmenten ist unser Feld `a` der Länge 100 vom Type `int` zugrundegelget und `i` sei eine einfache `int`-Variable:

- Alle Feldelemente mit dem Wert 1 versehen:

```
for ( i = 0; i < 100; i = i + 1 )
    a[i] = 1;
```
- Oder alle Feldelemente einlesen:

```
for ( i = 0; i < 100; i = i + 1 )
    scanf("%d", &a[i]);
```
- Oder alle Feldelemente in umgekehrter Reihenfolge ausgeben:

```
for ( i = 99; i >= 0; i = i - 1 )
    printf("%d ", a[i]);
```
- Oder nur jedes zweite Element ausgeben (vorwärts):

```
for ( i = 0; i < 100; i = i + 2 )
    printf("%d ", a[i]);
```

Es ist nicht zwangsläufig so, dass Felder “groß” sein müssen, es kann durchaus sinnvolle Anwendungen von Feldern der Länge 2 geben, etwa bei der Bearbeitung von Bildschirmpunkten:

Ein Bildschirmpunkt kann durch seine (ganzzahligen) Abstände vom linken und vom oberen Bildschirmrand beschrieben werden, die Definition `int punkt[2]`; wäre also durchaus sinnvoll.

In `felder/einleit/fibonac1.c` steht ein Beispiel, in dem in einem Feld die ersten 50 Glieder der durch $a_0 = 0$, $a_1 = 1$ und $a_i = a_{i-1} + a_{i-2}$ für $i \geq 2$ rekursiv definierten *Fibonacci-Folge* berechnet werden und gewisse dieser Glieder ausgeben werden können.

Bei der Definition eines Feldes muss die Feldlänge eine Konstante sein! (Genauer: ein konstanter Ausdruck — ein Ausdruck, dessen Wert beim Übersetzten schon feststeht. Variablen dürfen nicht in diesem Ausdruck vorkommen!)

Im obigen Beispiel ist diese Feldlänge 50 und diese Konstante 50 taucht an mehreren Stellen im Programmtext auf, etwa als obere Grenzen bei Schleifen bzw. bei Abfragen (immer dort, wo der Programmierer selbst dafür sorgen muss, dass nur zulässige Feldindizes verwendet werden!).

Ein Ändern dieser Feldlänge ist machbar, aber mühsam: Hier muss man den ganzen Quelltext durchgehen und jede auftretende Konstante 50 durch den gewünschten neuen Wert ersetzen! Schwierig wird es, wenn der Quelltext einige Dutzend Seiten lang ist und im Programm auch Konstanten 50 vorkommen, welche mit dem Feld nichts zu tun haben! Hier muss man bei jeder auftretenden 50 überprüfen, ob diese nun mit der Feldlänge zusammenhängt oder nicht!

Um (u.a.) diese Schwierigkeiten zu vermeiden, stellt der Präprozessor sog. *symbolische Konstanten* zur Verfügung. Die entsprechende Präprozessoranweisung (ganze Zeile) sieht dann so aus:

```
#define FELDLAENGE      50
```

Hinter dem `#define` steht zunächst ein Bezeichner (traditionell in Großbuchstaben)

und dahinter ein beliebiger Text (*Ersetzungstext*).

Überall, wo dieser Bezeichner im folgenden C-Quelltext als eigenständiges *Token* auftritt (also nicht als Teil eines anderen Bezeichners und nicht innerhalb von konstanten Zeichenketten), wird dieser Bezeichner vom Präprozessor (vor dem eigentlichen Übersetzen) durch den Ersetzungstext ersetzt!

Beim Programmieren verwendet der Programmierer dann überall, wo er sich auf die Feldlänge 50 bezieht, nicht die Konstante 50 sondern diese symbolische Konstante **FELDLAENGE**. Vor dem Übersetzen setzt dann der Präprozessor hierfür jeweils 50 ein und dem Programm liegt die gewünschte Feldlänge zugrunde!

Soll dann später mit einer anderen Feldlänge gearbeitet werden, so braucht nur die eine Präprozessoranweisung geändert werden, etwa:

```
#define FELDLAENGE      100
```

und nach einer erneuten Übersetzung des Programms wird mit der neuen Feldlänge gearbeitet. (Ein Überarbeiten des ganzen Quelltextes ist nicht mehr erforderlich!)

Die Verwendung dieses Konzeptes ist in `felder/einleit/fibonac2.c` demonstriert. (Auch das bereits wiederholt aufgetretene EOF für die Dateiendekennung ist eine symbolische Konstante, welche vom System in `stdio.h` vordefiniert ist.)

In Verzeichnis `felder/einleit` stehen weitere kleine Beispielprogramme zur Verwendung von Feldern.

`ziffern.c` ist ein Programm, welches in der Eingabe alle Zeichen, alle Zwischenraumzeichen (Leerzeichen, Tabulatoren und Zeilenvorschübe) zählt und zu jeder Ziffer feststellt, wie oft diese vorkommt. Der Zähler für die Ziffern ist hier als Feld realisiert.

In `abstand.c` wird der Abstand zweier (einzulesender) Punkte eines zweidimensionalen Koordinatensystems berechnet. Die beiden Punkte sind jeweils als Felder der Länge zwei vom Typ `double` realisiert: erstes Feldelement (Index 0) ist die x-Koordinate und das zweite Feldelement (Index 1) ist die y-Koordinate.

Zur Berechnung der Quadratwurzel wird hier die Bibliotheksfunktion

```
double sqrt(double);
```

verwendet.

Die Standardbibliothek stellt eine Reihe von mathematischen Funktionen zur Verfügung, zu deren Verwendung diese Header-Datei `math.h` mittels `#include` in den Quelldext eingebunden (und auf UNIX-Systemen beim Compilieren die Option `-lm` angegeben) werden muss.

5.1.1 Explizite Initialisierung von Feldern

Wie bei gewöhnlichen Variablen wird bei der Definition von Feldern entsprechend viel bis dahin freier Speicherplatz reserviert, ohne sich um den vorherigen Inhalt dieses Speicherbereiches zu kümmern. D.h. die einzelnen Feldelemente haben zunächst einen zufälligen Wert und man muss ihnen erst etwas zuweisen, damit man vernünftig mit ihnen arbeiten kann!

Wie bei gewöhnlichen Variablen können aber auch bei Feldern bei deren Vereinbarung initialisierende Werte angegeben werden. Dies erfolgt durch ein Gleichheitszeichen,

einer geschweiften Klammer auf, einer Liste von Werten des entsprechenden Types und der geschweiften Klammer zu, etwa:

```
int a[3] = { 5, 4, 3 };
```

es wird ein Feld vom Typ `int` der Länge 3 definiert, wobei das erste Feldelement `a[0]` den Wert 5 erhält, das zweite `a[1]` den Wert 4 und das dritte `a[2]` den Wert 3.

Hierbei darf die Länge der Liste der initialisierenden Werte kürzer sein als die definierte Feldlänge — die initialisierenden Werte beziehen sich dann auf die ersten Feldelemente, die restlichen werden nicht initialisiert, etwa:

```
int a[100] = { 5, 4, 3 };
```

es wird ein Feld der Länge 100 definiert, dessen ersten drei Elemente die Werte 5, 4 bzw. 3 und die restlichen 97 Elemente zufälligen Wert haben.

Die Liste der initialisierenden Ausdrücke darf nicht länger als die bei der Definition angegebene Feldlänge sein und es können keine Wiederholungsfaktoren (wie etwa: “100 mal der Wert ...”) formuliert werden!

Wird ein Feld bei seiner Definition gleich explizit initialisiert, so braucht in den eckigen Klammern keine Feldlänge angegeben zu werden (die eckigen Klammern selbst sind erforderlich). In diesem Fall wird die Feldlänge implizit vom Compiler anhand der Anzahl der initialisierenden Werte bestimmt, etwa:

```
double f[] = { 1.0, 4.6E2, -1.2345 };
```

das Feld `f` vom Typ `double` hat die Länge 3 (weil drei initialisierende Werte in der Liste stehen) und diese drei Feldelemente haben die angegebenen Werte.

5.2 Zeichenfelder

Felder vom Typ `char` sind zunächst wie jedes andere Feld zu verwenden: durch

```
char w[100];
```

werden 100 Bytes für 100 Zeichen-Variablen reserviert, die erste dieser Variablen hat den Namen `w[0]` und die letzte heisst `w[99]` und jede dieser Variablen kann man wie jede andere Variable vom Typ `char` verwenden.

Häufig sind jedoch in solchen Zeichenfeldern ganze Worte oder Sätze zu verarbeiten. Insbesondere kann es sein, dass in einem Zeichenfeld, etwa der Länge 100, einmal ein Satz bestehend aus 80 Zeichen und ein andermal nur ein Wort aus 5 Zeichen abgespeichert ist. Bei der “Ausgabe” des Feldes sollten dann das eine mal eben 80 Zeichen und das andere mal nur 5 Zeichen ausgegeben werden.

Neben der Feldlänge des Zeichenfeldes ist deshalb ebenfalls wichtig, wie lang der im Augenblick in dieser Zeichenkette abgespeicherte *String* (Wort oder Satz) ist.

Um keine separate Variable für diese “Belegungsgröße” verwalten zu müssen, gibt es in C folgende Konvention:

Ist in einem Feld vom Typ `char` ein Wort oder Satz (allg. *String* genannt) abgespeichert, so steht hinter dem letzten Zeichen des Strings das (ASCII-) Zeichen mit dem Wert 0 (C-Notation: `'\0'`, *Nullzeichen* oder *Stringendezeichen* genannt).

Ist also in unserem Zeichenfeld `char w[100]`; der String "hallo" abgespeichert, so sieht das etwa wie folgt aus:

w[0]	w[1]	w[2]	w[3]	w[4]	w[5]	w[6]	w[7]	w[8]		w[98]	w[99]
'h'	'a'	'l'	'l'	'o'	'\0'				...		

$\underbrace{\hspace{15em}}_w$

Die in `w[6]`, `w[7]`, ..., `w[99]` noch stehenden Zeichen (die Bit-Kombinationen in diesen Bytes werden auf jeden Fall als Zeichen interpretiert!) sind für den in `w` abgespeicherten String unerheblich, da sie ja hinter dem Stringendezeichen `'\0'` stehen.

Dies hat zur Folge, dass ein String in einem Zeichenfeld immer ein Byte mehr benötigt als seine eigentliche Länge und dass, wenn man einen String der Länge n in einem Feld abspeichern will, dieses Feld mindestens die Länge $n + 1$ haben muss!

Zur Stringbehandlung existieren in der Standardbibliothek eine Reihe von Funktionen, welche mit solchen variablen Zeichenketten (Feldern vom Typ `char`) arbeiten und all diese Funktionen verlassen sich auf diese Konvention. Sollte ein String durch eine Bibliotheksfunktion erzeugt werden, so sorgt diese Bibliotheksfunktion automatisch dafür, dass das Stringendezeichen angehängt wird. Erzeugt man selbst in einem Zeichenfeld einen String, so muss man auch selbst sicherstellen, dass hinten das Stringendezeichen angehängt wird:

```
char w[100];
...
w[0] = 'h';
w[1] = 'a';
w[2] = w[3] = 'l';
w[4] = 'o';
w[5] = '\0';      /* Stringendezeichen anhaengen ! */
```

Dies bezieht sich auch auf eine eventuelle explizite Initialisierung:

```
char wort[] = { 'h', 'a', 'l', 'l', 'o', '\0' };;
```

Alternativ kann hierfür auch

```
char wort[] = "hallo";
```

geschrieben werden, wobei der Compiler hier selbst für das `'\0'` sorgt.

(In beiden Fällen hat das Zeichenfeld die Länge 6!)

Das Stringendezeichen kann bei der Bearbeitung von Strings in natürlicher Weise verwendet werden. Zur Längenbestimmung eines in einem Zeichenfeld abgespeicherten Strings kann beispielsweise wie folgt vorgegangen werden:

```
int i;
char w[100];
...
for ( i = 0 ; w[i] != '\0' ; i = i + 1 )
    ;          /* leerer Anweisungsteil !!!! */
printf("Die Laenge der Zeichenkette w ist %d\n", i);
...
```

5.2.1 Ausgabe von Strings

Natürlich kann man mit einer Schleife einen String zeichenweise mit `putchar(c)` ausgeben. (Ende der Schleife beim Stringendezeichen!)

Einfacher geht es mit der `printf`-Funktion und der Ausgabespezifikation `%s`:

```
printf("%s", w);
```

gibt den Inhalt der Zeichenkette `w` aus, das den String terminierende Zeichen `'\0'` wird hierbei nicht ausgegeben! Selbst, wenn die Länge des Zeichenfeldes `w` mit 100 definiert ist, in diesem aber (wie oben im Beispiel) nur der String `"hallo"` abgespeichert ist, wird durch diesen `printf`-Aufruf nur das Wort `hallo` ausgegeben (5 Zeichen)!

Bei der Ausgabespezifikation `%s` kommt es dabei nicht darauf an, ob das zugehörige Argument eine (variables) Zeichenfeld (etwa `char w[100]`) oder eine konstantes Zeichenfeld ist, folgendes ist also möglich:

```
printf("Die Zeichenkette ist: %s\n", w);           bzw.  
printf("Die Zeichenkette ist: %s\n", "hallo");
```

(Intern werden konstante Zeichenketten auch als (konstante) Felder vom Typ `char` abgespeichert und auch diese Felder haben ein `'\0'` am Ende, so dass die konstante Zeichenkette `"hallo"` tatsächlich 6 Byte im Speicher belegt. Im Gegensatz zu variablen Zeichenfeldern können diese Bytes aber nicht im Programm verändert werden!)

Bei der Ausgabespezifikation `%s` kann wiederum eine (Ausgabe-)Feldbreite (etwa `%10s`, Mindestzahl auszugebender Zeichen ggf. wird vorne mit Leerzeichen aufgefüllt) eine Präzision (etwa `%.7s`, Höchstzahl der von der Zeichenkette auszugebenden Zeichen) oder auch beides (etwa `%10.7s`) angegeben werden.

Sofort hinter dem Prozentzeichen kann im Zusammenhang mit einer Feldbreite als weitere Steuerflagge ein Minuszeichen angegeben sein. Dies bewirkt eine linksbündige Ausgabe innerhalb der angegebenen Feldbreite, d.h. es wird ggf. hinten mit Leerzeichen aufgefüllt.

Analog zur Funktion `int putchar(char);` zur Ausgabe einzelner Zeichen gibt es eine Funktion

```
int puts(char a[]);
```

zur Ausgabe von Zeichenketten. (Die eckigen Klammern im Argument `char a[]` bedeuten, dass das Argument `a` ein Feld vom angegebenen Typ sein muss, also hier ein Feld vom Typ `char`!) Ist `w` ein Feld vom Type `char`, so ist der Aufruf wie folgt:

```
puts(w);
```

Die Funktion gibt die übergebene Zeichenkette und anschließend ein `'\n'` aus. Das Funktionsergebnis (muss nicht verwendet werden) ist größer als 0 bzw. `EOF` im Fehlerfall!

Auch diese Funktion kann mit konstanten Zeichenketten als Argument aufgerufen werden:

```
puts("hallo");
```

5.2.2 Eingabe von Strings

Ist `w` ein Feld vom Typ `char`, so kann dieses mit dem Aufruf

```
scanf("%s", w);
```

eingelezen werden. (Man beachte, dass vor dem Feldnamen kein `&`-Zeichen steht!!!) Durch diesen Aufruf werden zunächst führende Zwischenraumzeichen (Leerzeichen, Tabulatoren und Zeilenvorschübe) auf dem "Eingabefließband" überlesen und die dann folgenden Zeichen (bis ausschließlich des nächsten Zwischenraumzeichens) gelesen und der Reihe nach den Elementen des Zeichenfeldes `w` zugewiesen. Dieses Lesen endet, wie erwähnt, mit dem nächsten Zwischenraumzeichen, welches dann auf dem Eingabefließband stengelassen wird! Hinter dem zuletzt gelesenen und im Feld `w` abgespeicherten Zeichen wird hierbei in `w` das Stringendezeichen `'\0'` angehängt! (D.h. `scanf` beachtet die Konvention für Strings!)

Die Funktion `scanf` geht beim Lesen von Strings davon aus, dass die als Argument übergebene Zeichenkette lang genug ist, um die eingelesenen Zeichen und das anschließende `'\0'` abzuspeichern! (D.h. in `scanf` wird kein Feldüberlauf abgefangen!) Auch hier hat der Programmierer selbst dafür zu sorgen, dass die übergebenen Zeichenfelder lang genug sind (`'\0'` berücksichtigen), um die Eingabe korrekt aufnehmen zu können!

Alternativ zur Eingabespezifikation `%s` für Zeichenketten gibt es folgende: `%[...]`, wobei `...` eine Liste von unmittelbar hintereinander stehenden Zeichen ist. (Dies ist die Liste der bei diesem Lesebefehl "akzeptablen" Zeichen!)

Diese Eingabespezifikation liest die nächsten "akzeptablen" Zeichen von der Eingabe und speichert sie der Reihe nach im übergebenen Feld vom Typ `char` ab. Beim nächsten "nicht akzeptablen" Zeichen in der Eingabe endet der Lesevorgang und dieses Zeichen bleibt auf der Eingabe stehen und an das letzte im Feld abgespeicherte Zeichen wird `'\0'` angehängt. Das Einlesen ist nur dann "erfolgreich" falls mindestens ein Zeichen gelesen und zugewiesen wurde! Ansonsten endet der `scanf`-Aufruf mit einem Funktionsergebnis, welches diese Nichtzuweisung deutlich macht!

Beispiel:

```
scanf("%[AEIOUaeiou]", w);
```

liest von der Eingabe, solange dort nur Vokale (kleine oder große) stehen. Beim nächsten Nichtvokal endet der Lesevorgang.

Bei derartigen Eingabespezifikationen werden führende Zwischenraumzeichen nicht überlesen, sondern bewirken, falls sie nicht selbst "akzeptabel" sind (d.h. innerhalb der eckigen Klammern aufgeführt sind) das Ende des Lesevorgangs.

Zur Abkürzung der Liste der "akzeptablen" Zeichen können in den eckigen Klammern mit einem Minuszeichen *Zeichenbereiche* angegeben werden, etwa `%[A-Z]` für alle Großbuchstaben oder `%[A-Za-z]` für alle Buchstaben.

Soll das Zeichen `]` selbst "akzeptabel" sein, so ist dies durch `%[...]` anzugeben.

Soll das Zeichen `-` selbst "akzeptabel" sein, so ist dies durch `%[...-]` anzugeben.

Ist `^` das erste Zeichen hinter der eckigen Klammer in einer solchen Eingabespezifikation, so wird die Menge der "akzeptablen" Zeichen komplementiert, d.h. alle Zeichen

außer den angegebenen sind “akzeptabel“ (d.h. werden gelesen und zugewiesen), etwa: `%[^0-9]` liest alle Zeichen, welche keine Ziffern sind!

Bei Einlesevorgängen mit `%s` bzw. `%[...]` kann wiederum eine Feldbreite angegeben werden (etwa: `%10s` oder `%8[...]`). Die angegebene Zahl ist dann die Höchstzahl der einzulesenden Zeichen (wobei beim Umwandlungszeichen `s` führende Zwischenraumzeichen nicht mitgezählt werden!)

Ist bei der Einlesespezifikation ein `*` angegeben, so wird wie üblich gelesen, aber eine Zuweisung unterbunden.

Bei `scanf("%*[\t\n] %[A-Za-z\t]", w);` etwa geschieht folgendes: zunächst werden führende Zwischenraumzeichen (es muss mindestens eins vorliegen) überlesen, anschließend wird die längste Zeichenkette (darf nicht die Länge 0 haben!), welche nur aus Groß- bzw. Kleinbuchstaben bzw. Leerzeichen bzw. Tabulatoren besteht, gelesen und (mit einem abschließenden `'\0'`) dem Feld `w` zugewiesen!

Analog zur Funktion `int getchar(void);` zum Einlesen eines einzelnen Zeichens gibt es die Funktion `gets(char w[]);`, welche von der Eingabe alle folgenden Zeichen bis (einschließlich) des nächsten `'\n'` liest (und dabei vom Eingabefließband entfernt), und die gelesenen Zeichen (ausschließlich des am Ende gelesenen `'\n'`) der Reihe nach im übergebenen Feld abspeichert und schließlich ein `'\0'` anhängt. (Auch diese Funktion hat ein Funktionsergebnis, welches wir aber erst später behandeln können!)

5.2.3 Weitere Bibliotheksfunktionen zur Stringbearbeitung

Zur Verwendung weiterer Funktionen zur Stringbearbeitung muss die Header-Datei `string.h` eingebunden werden. Es stehen dann (u.a.) folgende Funktionen zur Verfügung:

`strcpy(char to[], char from[]);`

kopiert den Inhalt der Zeichenkette in `from` (bis einschließlich `'\0'`) nach `to`. (Das Feld `to` muss lang genug sein!)

`strcat(char to[], char from[]);`

hängt den Inhalt der Zeichenkette `from` an die Zeichenkette `to` an. (Das Stringendezeichen von `to` wird durch das erste Zeichen von `from` ersetzt, die weiteren Zeichen bis einschließlich dem Stringendezeichen von `from` werden angehängt. Das Feld `to` muss lang genug sein!))

`int strcmp(char a[], char b[]);`

vergleicht die beiden in `a` bzw. `b` stehenden Zeichenketten lexikographisch anhand des Maschinenzzeichensatzes (i. Allg. kommen alle Großbuchstaben vor den Kleinbuchstaben, somit kommt ein `'Z'` vor einem `'a'`). Würde die in `a` abgespeicherte Zeichenkette im Lexikon vor der in `b` abgespeicherten stehen, so ist das Funktionsergebnis kleiner als 0, sind beide Zeichenketten identisch, so ist das Funktionsergebnis gleich 0 und ansonsten größer als 0.

`int strlen(char a[]);`

gibt als Funktionsergebnis die Länge der als Argument übergebenen Zeichenkette (gleich Anzahl der Zeichen ohne das Stringendezeichen!) zurück.

Das Objekt `c` kann als ein Feld der Länge 2 aufgefasst werden, wobei jedes Feldelement von `c` ein zweidimensionales Feld des oben beschriebenen Typs `double a[3][5]` ist. Insgesamt liegen $2 \cdot 3 \cdot 5 = 30$ Elemente vom Typ `double` zugrunde und der Zugriff auf diese `double`-Elemente erfolgt über drei Indizes: `c[i][j][k]`, wobei `i` aus dem Bereich von 0 bis 1, `j` aus 0 bis 2 und `k` aus 0 bis 4 sein muss.

Zweidimensionale Felder vom Typ `int` oder `double` werden häufig zur Lösung mathematischer Probleme verwendet — sie heißen dort *Matrizen*.

Zweidimensionale Felder vom Typ `char` sind ebenfalls nichts ungewöhnliches, etwa `char tab[20][32]`;

Dieses Objekt `tab` besteht aus 20 Zeichenfeldern der Länge 32 — kann also zur Abspeicherung von 20 Worten, welche jeweils höchstens die Länge 31 haben dürfen (Stringdezeichen beachten!), verwendet werden. Somit realisiert dieses zweidimensionale Feld vom Typ `char` ein eindimensionales Feld von Strings (Worten)!

Die naheliegende Bedeutung von `tab[i]` als $(i - 1)$ -tes Element des eindimensionalen Feldes von Strings wird von C unterstützt: überall, wo auch der Name eines Feldes vom Typ `char` verwendet werden könnte, kann etwa auch `tab[0]` bis `tab[19]` auftreten, etwa:

- `scanf("%s", tab[2]);`
es wird eine Zeichenkette von der Tastatur gelesen und die gelesene Zeichenkette wird als dritte Zeile in (dem zweidimensionalen Feld) `tab` abgespeichert. (Die gelesene Zeichenkette sollte hierbei höchstens die Länge 31 haben!)
- `strcpy(tab[19], "Hanrath");`
in die letzte Zeile von (dem zweidimensionalen Feld) `tab` wird die Zeichenkette "Hanrath" abgespeichert.
- ...

Die Belegung dieses Feldes `tab` könnte etwa so aussehen (und hierbei wird die Bedeutung "eindimensionales Feld von Worten" nochmals deutlich!):

tab[0]:	'H'	'a'	'n'	'r'	'a'	't'	'h'	'\0'	...		
tab[1]:	'S'	'c'	'h'	'u'	'l'	't'	'z'	'\0'	...		
tab[2]:	'D'	'i'	'e'	't'	'e'	'l'	'\0'		...		
tab[3]:	'B'	'r'	'e'	'u'	'e'	'r'	'\0'		...		
⋮											
tab[18]:	'K'	'e'	'l'	'l'	'e'	'r'	'\0'		...		
tab[19]:	'H'	'o'	'f'	'f'	'm'	'a'	'n'	'n'	'\0'	...	

Im Zusammenhang mit einfachen Feldern ist die Problematik des Sortierens eines

Feldes behandelt worden, und zwar anhand eines Feldes vom Typ `int` (vgl. `felder/einleit/sort.c`).

Das Sortieren eines Feldes von Worten (also eines zweidimensionalen Feldes vom Typ `char`) kann völlig analog durchgeführt werden. Es muss nur beachtet werden, dass

- zum direkten Vergleich zweier Worte nicht der Vergleichsoperator `<` verwendet werden kann (dieser ist nicht für Zeichenketten definiert), sondern die Bibliotheksfunktion `strcmp` und
- die Zuweisung bei einer notwendigen Vertauschung zweier Strings nicht durch den Zuweisungsoperator `=` erfolgen kann (dieser ist ebenfalls nicht für Felder definiert!), sondern hier ebenfalls die entsprechende Bibliotheksfunktion `strcpy` verwendet werden muss!

Das Sortieren eines derartigen Feldes von Wort ist in `felder/char/namesort.c` demonstriert.

5.3.1 Initialisierung mehrdimensionaler Felder

Wie eindimensionale Felder, können auch mehrdimensionale Felder bei ihrer Definition gleich initialisiert werden. Hierzu muss dem Gleichheitszeichen hinter der Dimensionierung eine in geschweifte Klammern stehende Liste von *initialisierenden Ausdrücken* folgen, wobei jeder dieser initialisierenden Ausdrücke ein eindimensionales Feld (Zeile) initialisiert, etwa:

```
int a[2][3] = { {1,2,3}, {4,5,6} };
```

Die erste (und nur die erste) Dimensionsangabe darf hierbei fortgelassen werden (hier also die 2, die eckigen Klammern müssen stehen!) — diese Dimensionierung wird vom Compiler anhand der Anzahl der initialisierenden Ausdrücke ermittelt! Folgende Initialisierung wäre also vollkommen gleichwertig zur obigen:

```
int a[][3] = { {1,2,3}, {4,5,6} };
```

Die initialisierenden Ausdrücke, also jeweils Listen von Werten zur Initialisierung eines eindimensionalen Feldes, brauchen nicht alle gleich lang zu sein, dürfen aber höchstens so lang sein, wie die zweite Dimensionierung angibt! Entsprechend nicht initialisierte Feldelemente werden dann eben nicht explizit vorbesetzt!

Die explizite Initialisierung von höherdimensionalen Feldern erfolgt analog, etwa:

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

zur Initialisierung eines Feldes, dessen Elemente selbst zweidimensionale Felder mit 2 Zeilen und 3 Spalten sind.

Auch hier kann nur die erste Dimensionsangabe fortgelassen werden (die eckigen Klammern wiederum nicht) — diese wird anhand der Anzahl der initialisierenden Ausdrücke vom Compiler ermittelt (hier ist diese also 2!).

Bei der Initialisierung mehrdimensionaler Felder vom Typ `char` muss wiederum das Stringendezeichen `\0` bedacht werden, etwa:

```
char tab[][16] = { { 'h', 'a', 'l', 'l', 'o', '\0' }, { 'w', 'e', 'l', 't', '\0' } };
```

Ersatzweise könnte man hierfür wiederum:

```
char tab[][16] = { "hallo", "welt" };  
schreiben.
```

6 Funktionen

6.1 Grundlagen zu Funktionen

Wir haben bereits Funktionen aus der Standardbibliothek kennengelernt und in unseren Programmen verwendet, etwa die Funktionen `printf`, `scanf`, `strcpy`,

Diese erledigen gewisse (umfangreiche) Aufgaben. Sie können an unterschiedlichen Stellen eines Programms aufgerufen werden, die Funktion erledigt die ihr zugedachte Aufgabe und anschließend fährt das Programm hinter dem Aufruf der Funktion fort. Der Aufruf der Funktion erfolgt durch Angabe des Funktionsnamens, gefolgt von einer runden Klammer auf, ggf. einer Liste von passenden Argumenten, welche zur korrekten Ausführung der Funktion erforderlich sind, und der schließenden runden Klammer, etwa:

```
- printf("Das Ergebnis ist: %d\n", i);      oder
- strcpy(a, "Hund");
```

Gewisse Funktionen liefern ein Funktionsergebnis, welches in Ausdrücken weiter verwendet werden kann, etwa:

```
- c = getchar()
- if ( (c = getchar()) != EOF) ...
- if ( strcmp(a,b) > 0 )
```

Damit der Compiler beim Übersetzen feststellen kann, ob eine Funktion korrekt aufgerufen wurde (ob Typ und Anzahl der Funktionsargumente stimmt und ob ggf. das Funktionsergebnis in den Zusammenhang des Funktionsaufrufs passt), muss dem Compiler die Funktion bekannt sein, bevor sie aufgerufen wird. Diese Bekanntmachung (Deklaration) der Funktionen der Standardbibliothek erfolgt in den Bibliotheks-Header-Dateien (`stdio.h`, `string.h`, `math.h` ...), welche dann ggf. durch die entsprechende Präprozessoranweisung einzubinden sind!

Diese Bekanntmachung umfasst den Namen der Funktion und den *Typ der Funktion*. Zum Typ der Funktion gehört

- der *Rückgabotyp* (Ergebnistyp): das ist der Typ des Funktionsergebnisses. Dieser Rückgabotyp entscheidet darüber, wie die Funktion innerhalb komplexerer Ausdrücke aufgerufen werden kann.

Eine Funktion, welche ein Funktionsergebnis vom Typ `int` hat (z.B. die Funktion `strlen`!), kann etwa als Index bei einem Feldzugriff verwendet werden:

```
y = a[ strlen(a) ];
```

Ein derartiger Aufruf mit der Funktion `sqrt`: `y = a[ sqrt(25.0) ]`; ist jedoch syntaktisch falsch, da das Funktionsergebnis dieser Bibliotheksfunktion `sqrt` vom Typ `double` ist und `double`-Werte als Feldindizes nicht zulässig sind!

- die *Signatur der Funktion*: das ist Typ, Anzahl und Reihenfolge der für diese Funktion notwendigen Argumente.

Anhand der Signatur erkennt der Compiler, ob eine Funktion sachgerecht aufgerufen wurde.

Die Funktion `sqrt` etwa benötigt einen double-Wert als Argument. Sind Variablen wie folgt definiert: `double a, b[10]`;, so ist der Aufruf `sqrt(a)`; syn-

taktisch korrekt, der Aufruf `sqrt(b)`; jedoch nicht, da `b` ein `double`-Feld ist und somit einen falschen Typ hat.

Falls möglich behilft sich der Compiler hier mit Typangleichungen, z.B. ist der Aufruf `sqrt(i)`; mit einer `int`-Variablen `i` korrekt! Die Funktion `sqrt` verlangt ein `double`-Argument — das hier jedoch vorliegende `int`-Argument wird (da klar ist, wie dies zu geschehen hat) beim Aufruf von `sqrt` automatisch in einen `double`-Wert umgewandelt!

Zur Erledigung gewisser Aufgaben kann man auch selbst Funktionen schreiben und diese an unterschiedlichen Stellen des Programms aufrufen.

Der Sinn derartiger selbst geschriebener Funktionen (wie auch der der Bibliotheksfunktionen) ist es:

- Wohlabgeschlossene Teilaufgaben, welche u. U. mehrfach im Programm erledigt werden müssen, nur einmal programmieren zu müssen und ggf. an unterschiedlichen Stellen im Programm, überall wo es erforderlich ist, aufrufen zu können.
- Das Programm zu strukturieren.

Eine wohlabgeschlossene Teilaufgabe, auch wenn sie nur einmal zu erledigen ist, kann durchaus sinnvoll als Funktion realisiert werden. (Das u. U. umfangreiche Problem, welches mit dem Programm zu lösen ist, wird in einzelne, überschaubare Teilprobleme aufgeteilt und die Lösung jedes der Teilprobleme wird dann jeweils durch eine einzelne, überschaubare Funktion erledigt.)

Als Beispiel für ein Teilproblem soll hier die ganzzahlige Potenzierung dienen: es sollen Potenzen der Form 5^3 oder i^5 oder i^j (mit ganzzahligen Größen i und j) berechnet werden. (In C gibt es keinen entsprechenden Operator, die Bibliotheksfunktion `double pow(double, double)`; ist für `double`-Werte gedacht!)

Die *Signatur* der zu schreibenden Funktion (sie soll den Namen `potenz` erhalten) liegt somit schon fest: sie benötigt zwei ganzzahlige Argumente (Typ `int`), die *Basis* und den *Exponenten*.

Auch der Rückgabetyt dieser Funktion ist klar: das Ergebnis soll ebenfalls vom Typ `int` sein!

Diese Information: Name, Signatur und Rückgabetyt der Funktion ist für den Anwender ausreichend, um die Funktion korrekt aufrufen zu können und diese Information reicht dem Compiler, um die Korrektheit des Aufrufs zu überprüfen. Genau diese Information wird dem Compiler vor der Verwendung (Aufruf) der Funktion durch die sog. Funktionsdeklaration mitgeteilt. Diese hat die Form:

```
int potenz(int, int);
```

und kann überall dort stehen, wo üblicherweise Variablen definiert werden, also am Anfang des Programms, hinter der geschweiften Klammer auf vor den ersten Anweisungen des Programms! An dieser Deklaration ist der Rückgabetyt (`int` vor dem Namen der Funktion), der Name der Funktion (`potenz`) und die Signatur (Anzahl und Typ der Argumente, hier zwei Argumente vom Typ `int`) ersichtlich!

Anschließend kann die so deklarierte Funktion im Programm verwendet werden, etwa:

```
j = potenz(i,5);
```

wobei der Compiler durch die ihm bekannte Deklaration dieser Funktion die Möglich-

keit hat, die Korrektheit der Aufrufe zu überprüfen und ggf. eine Fehlermeldung auszugeben, etwa bei folgendem falschen Aufruf:

```
j = potenz(i); /* Fehler: Es fehlt ein Argument!*/
```

(In den Header-Dateien der Standardbibliothek stehen — zwar an anderer Stelle — genau diese Deklarationen der entsprechenden Funktionen der Standardbibliothek. Nach dem Einbinden der Header-Dateien sind dem Compiler diese Funktionen bekannt!)

Im Gegensatz zu den Funktionen der Standardbibliothek müssen eigene Funktionen auch definiert werden.

Die Definition einer eigenen Funktion erfolgt (im einfachsten Fall) hinter der schließenden geschweiften Klammer des Hauptprogramms!

Die erste Zeile der Funktionsdefinition ist ähnlich zu der Deklaration der Funktion — nur der Strichpunkt am Ende fehlt und bei den Parametertypen innerhalb der runden Klammern stehen hinter den Typennamen jeweils (Variablen-)Namen, also Bezeichner (Anzahl und Typ der Parameter muss bei Deklaration und Definition gleich sein!):

```
int potenz(int a, int b)
```

Hinter dieser ersten Zeile der Funktionsdefinition steht der *Anweisungsteil* der Funktion, d.h. es folgt eine geschweifte Klammer auf, anschließend können weitere Variablen definiert werden, hinter den Variablendefinitionen folgen Anweisungen und hinter den Anweisungen folgt die schließende geschweifte Klammer. D.h. der Anweisungsteil der eigenen Funktion ist vollkommen analog zum dem des Hauptprogramms **main**!

(Strenggenommen ist **main** auch “nur“ eine selbst geschriebene Funktion, welche (bislang) kein Funktionsergebnis liefert — deshalb das Schlüsselwort **void** vor dem Funktionsnamen — und (bislang) auch keine Argumente benötigt — deshalb das Schlüsselwort **void** in den runden Klammern! Die Besonderheit von **main** ist, dass beim Programmstart vom System eben diese “eigene“ Funktion mit dem Namen **main** aufgerufen wird und dass, wenn diese Funktion beendet ist, der Programmablauf ebenfalls zuende ist und es im Betriebssystem weitergeht!)

Das Fragment zur Verwendung und Definition unserer Funktion sieht wie folgt aus (komplettes Beispiel: **funktion/potenz.c**):

```
void main(void)
{ int potenz( int, int); /* Deklaration der Funktion potenz */
  int i, j, erg;
  ...
  erg = potenz( i , j ); /* Aufruf der Funktion potenz */
  ...
  return;
}
```

```

/* Definition der Funktion potenz */
int potenz(int base, int exp)
{ /* Definition weiterer hier notwendiger Variablen */
    int i, pow;
    pow = 1;

    for ( i = 0; i < exp; i = i + 1)
        pow = pow * base;

    return pow;
}

```

Die in der ersten Zeile der Funktionsdefinition angegebenen Namen für die Argumente sind tatsächlich Namen von Variablen. Diese Variablen heißen *Funktionsparameter*. Im Anweisungsteil der Funktion stehen weitere, “gewöhnliche“ Variablendefinitionen, hier: `int i, pow;`.

Beim Aufruf der Funktion werden die entsprechenden Variablen (Funktionsparameter und “gewöhnliche“ Variablen) neu angelegt. Die Funktionsparameter erhalten dabei sofort einen Wert, nämlich den Wert des beim Aufruf der Funktion an entsprechender Stelle stehenden Argumentes (Funktionsargument).

Beim Aufruf `potenz(5,3)` erhält also der Parameter `base` den Wert 5 und der Parameter `exp` den Wert 3. (Beim Aufruf können als Argumente beliebige Ausdrücke stehen, welche einen Wert vom passenden Typ oder einen durch Standard-Typumwandlung in den passenden Type umwandelbaren Wert liefern, etwa `potenz( i - j, j + 5)`, falls `i` und `j` Variablen vom Typ `int` sind. Der Typ des Ausdrucks `i - j` ist `int` und der Wert dieses Ausdrucks wird bei diesem Aufruf der (Parameter-)Variablen `base` zugewiesen, der Ausdruck `j + 5` hat ebenfalls den richtigen Typen `int` und der Wert dieses Ausdrucks wird der (Parameter-)Variablen `exp` zugewiesen.)

Die sonstigen im Anweisungsteil einer Funktion neu definierten Variablen haben (falls sie nicht explizit initialisiert werden) einen zufälligen Wert!

Die Variablen der Funktion (Funktionsparametern und sonstiger Variablen) haben mit den Variablen der “Hauptfunktion“ `main` (oder allgemein: der aufrufenden Funktion), nichts zu tun. In der Hauptfunktion (`main`) kann nur auf Variablen der Hauptfunktion zugegriffen werden und in der aufgerufenen Funktion kann nur auf Variablen dieser aufgerufenen Funktion zugegriffen werden. (Variablen sind *lokal*!)

Selbst gleichnamige Variablen in aufrufender und aufgerufener Funktion (es gibt in unserem Beispiel in `main` eine `int`-Variable mit dem Namen `i` und in der Funktion `potenz` ebenfalls!) sind verschiedene Objekte, haben unterschiedliche Speicherbereiche und in der aufrufenden Funktion kann nur auf den in ihr für `i` reservierten Speicherbereich zugegriffen werden und in der aufgerufenen Funktion nur auf den in der Funktion definierten!

Nach der Definition der neuen Variablen werden beim Aufruf einer Funktion die Anweisungen dieser Funktion ausgeführt, bis der Programmfluß an der `return`-Anweisung

angelangt ist. Diese wird noch ausgeführt und anschließend geht der Programmfluss hinter dem Aufruf in der aufrufenden Funktion weiter. Bei diesem Ende des Funktionsaufrufes wird der Speicherbereich der Variablen der (gerade beendeten) Funktion wieder freigegeben, d.h. die zur Funktion lokalen Variablen existieren anschließend nicht mehr und werden ggf. bei einem nächsten Aufruf der Funktion wieder neu erzeugt!

In unserem Beispiel sehen wir eine neue Form der **return**-Anweisung:

```
return pow;
```

wobei **pow** eine Variable der Funktion ist, deren Wert die (in der Funktion durch eine Schleife berechnete) eigentlich gewünschte Potenz basis^{exp} ist. Das Funktionsergebnis der Funktion **potenz** soll ein Wert vom Typ **int** sein (erkennbar an Deklaration und Kopfzeile der Definition der Funktion)! Der hinter **return** stehende Ausdruck (hier eine einfache Variable) hat einen Wert vom entsprechenden Typ.

Der Wert des Funktionsaufrufes ist der Wert des Ausdrucks hinter **return** am Ende der Funktion! D.h. beim Aufruf **erg = potenz(i, j);** ist der Wert des Funktionsaufrufes **potenz(i, j)** der Wert der zur Funktion lokalen Variablen **pow** am Ende der Funktion und dieser Wert wird hier (im Hauptprogramm) der Variablen **erg** zugewiesen!

Hier ist zu beachten, dass bei der Zuweisung die Funktion bereits beendet ist und somit insbes. auch die funktionseigene Variable **pow** gar nicht mehr existiert! Vom System wird zur Zwischenspeicherung von Funktionsergebnissen ein temporärer Speicherbereich angelegt, in dem das Funktionsergebnis beim **return** abgelegt wird und der über das Funktionsende hinaus erhalten bleibt, bis das Funktionsergebnis in dem Ausdruck, in welchem die Funktion aufgerufen wurde, weiter verwendet worden ist! D.h. bei einem Aufruf:

```
erg = potenz ( ... , ... );
```

wird der Inhalt der funktionseigenen Variablen **pow** nicht direkt der Variablen **erg** des aufrufenden Programmteils zugewiesen, sondern über den Umweg über einen temporären Zwischenspeicher!

Es ist durchaus möglich, dass in einer Funktion mehrere **return**-Anweisungen vorkommen:

```
int fkt(...)
{ int dies, jenes;
  double das;
  ...
  if ( Fall1 ) return dies;
  ...
  if ( Fall2 ) return das;
  ...
  return jenes; /* sonst */
}
```

Bei der Rückgabe eines Wertes wird ggf. wiederum eine Typangleichung vorgenommen: entspricht (wie im obigen im **Fall2**) der Typ des hinter **return** angegebenen

Ausdruckes (hier Typ `double`) nicht dem Ergebnistyp der Funktion (hier Typ `int`), so wird, falls es mit einer Standard-Typumwandlung möglich ist, der hinter `return` angegebene Wert in den Ergebnistyp der Funktion umgewandelt!

Ansonsten es ist der Verantwortung des Programmierers überlassen, vernünftige Rückgabewerte vom richtigen Typ zurückzugeben, insbesondere überhaupt ein Ergebnis zurückzugeben. Unsere Funktion `int fkt(...)` könnte auch einfach mit `return;` oder ohne `return`-Anweisung beendet werden (eine Funktion ist spätestens dann beendet, wenn der Programmfluss an der schließenden geschweiften Klammer des Anweisungsteils der Funktion angelangt ist!)! In diesen Fällen wird kein Funktionsergebnis explizit zurückgegeben — der Funktionsaufruf `fkt(...)` hat trotzdem einen Wert, nämlich einen zufälligen! (Dies ist i. Allg. nicht sinnvoll!)

Ein Wort noch zur *Schnittstelle* zu einer Funktion.

Die Schnittstelle zu einer Funktion ist die Information, welche wichtig ist, um die Funktion korrekt aufrufen und anwenden zu können.

Diese Information steht in der Funktionsdeklaration!

Hierin ist aufgeführt, welche Argumente beim Aufruf anzugeben sind und was für ein Funktionsergebnis herauskommt!

Beim Aufruf der Funktion springt der Programmfluss von dem aufrufenden Programmteil in die Funktion, d.h. es werden die Anweisungen der Funktion ausgeführt, und am Funktionsende springt der Programmfluss wieder ins aufrufende Programm hinter die Stelle, wo die Funktion aufgerufen wurde und es wird, u. U. unter Verwendung des Funktionsergebnisses, mit der Abarbeitung der dort stehenden Anweisungen fortgefahren (vgl. Abbildung 13 auf Seite 42)!

Eine Funktion bekommt i. Allg. also ihre Daten aus den Funktionsparametern und gibt ihr Ergebnis als Funktionswert zurück!

Es ist i. Allg. nicht sinnvoll, innerhalb der Funktion weitere neue Daten von Tastatur einzulesen oder Ergebnisse auf dem Bildschirm auszugeben!

Eingabe von Tastatur und Ausgabe auf den Bildschirm sollte entweder im Hauptprogramm oder in speziell dafür vorgesehenen Eingabefunktionen und Ausgabefunktionen erfolgen!

6.2 Funktionsargumente, Funktionsparameter, *Call by Value*

Funktionsparameter einer Funktion sind lokale Variablen der Funktion und beim Funktionsaufruf wird für diese (wie auch für andere lokale Variablen der Funktion) lokal neuer Speicherbereich reserviert!

Diese Parameter haben gegenüber den sonstigen lokalen Variablen der Funktion die Auszeichnung, dass ihnen bei Ihrer Erzeugung gleich ein Wert gegeben wird, nämlich der Wert des zugehörigen, beim Funktionsaufruf angegebenen Argumentes!

Im obigen Beispiel etwa sind `i` und `j` Variablen des Hauptprogramms und somit steht im Hauptprogramm Speicherbereich (u.a.) für diese beiden Variablen zur Verfügung. Im Hauptprogramm wird dann die Funktion `potenz` durch `potenz( i, j)` aufgerufen. Die Kopfzeile der Funktionsdefinition ist `int potenz (int base, int exp)`

und das bedeutet, dass jetzt bei diesem Aufruf mindestens zwei weitere Variablen **base** und **exp** vom Typ **int** erzeugt werden und dabei für diese (neuer) Speicherbereich reserviert wird. Diese beiden Funktionsparameter erhalten bei ihrer Erzeugung als Wert den Wert des korrespondierenden Argumentes (im aufrufenden Programmteil). D.h. die Variable **base** der Funktion **potenz** bekommt den gleichen Wert, den die Variable **i** des aufrufenden Programmteils beim Aufruf von **potenz** hat und die Variable **exp** der Funktion **potenz** bekommt den gleichen Wert, den die Variable **j** des aufrufenden Programmteils beim Aufruf von **potenz** hat.
(Deswegen: *Call by value* — es werden Werte übergeben!)

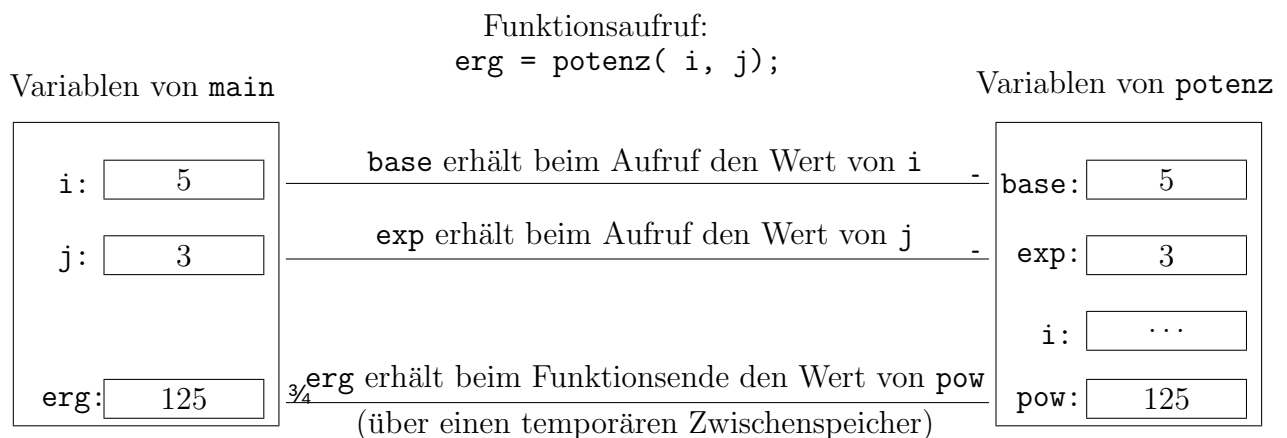


Abbildung 23: Datenfluss beim Funktionsaufruf

Nach dieser Wertzuweisung beim Aufruf der Funktion ist die Sonderrolle der Parametervariablen **base** und **exp** vorbei und sie sind nun ganz gewöhnliche lokale Variablen der Funktion! Insbesondere können sie innerhalb der Funktion verändert werden — die Funktionsargumente im aufrufenden Programmteil bleiben davon unberührt! Wird etwa innerhalb der Funktion **potenz** die Variable **exp** verändert, so hat das keinen Einfluss auf die Variable **j** des Hauptprogramms (diese ist beim obigen Aufruf das zum Funktionsparameter **exp** korrespondierende Funktionsargument!).

Dies kann bei der Definition der Funktion ausgenutzt werden, um etwa mit einer Variablen weniger auszukommen:

```
/* Definition der Funktion potenz */
int potenz(int base, int exp)
{ /* Definition und Initialisierung weiterer hier notwendiger Variablen */
    int pow = 1;
    /* Parameter exp wird als Schleifenvariable benutzt */
    /* Initialisierung entfaellt */
    for ( ; exp > 0 ; exp = exp - 1)
        pow = pow * base;

    return pow;
}
```

Hier wird der zweite Funktionsparameter `exp` als Schleifenvariable verwendet und beim Durchlaufen der Schleife entsprechend erniedrigt, so dass er nach der Schleife und auch beim Ende der Funktion den Wert 0 hat.

Wird die Funktion durch `erg=potenz( i, j);` aufgerufen und hat hierbei die Variable `j` (korrespondiert als Argument zum Parameter `exp`) vor dem Aufruf etwa den Wert 3, so hat sie nach dem Aufruf immer noch den Wert 3, d.h. die Änderung eines Funktionsparameters `exp` innerhalb der Funktion ist ohne Wirkung für das entsprechende Argument `j` in der aufrufenden Funktion!

6.3 Funktionen ohne Argumente

Es gibt Funktionen, die ohne Argumente auskommen und denen beim Aufruf auch keine Argumente mitgegeben werden können.

Bei derartigen Funktionen muss bei der Deklaration innerhalb der runden Klammern, in denen normalerweise die Liste der Typen der Parameter steht, das Schlüsselwort `void` stehen.

Entsprechend haben solche Funktionen auch keine Funktionsparameter und bei der Funktionsdefinition ist in der Kopfzeile ebenfalls innerhalb der runden Klammern dieses Schlüsselwort `void` anzugeben!

Beispiel: Eine Funktion mit dem Namen `getint`, welche eine ganze Zahl von der Tastatur liest und als Funktionsergebnis den Wert der gelesenen Zahl als `int` zurückgibt. Diese Funktion benötigt keine Argumente.

Die Deklaration könnte so aussehen:

```
int getint(void);
```

und die Definition könnte sein:

```
int getint(void)
{ int i;          /* lokale int Variable */
  scanf("%d", &i); /* lokale int Variable einlesen */
  return i;       /* gelesenen Wert als Funktionsergebnis
                  zurueckgeben */
}
```

Der Aufruf dieser Funktion (innerhalb einer anderen Funktion) könnte so aussehen:

```
...
int k;
...
k = getint();
...
```

Der Aufruf erfolgt also durch Angabe des Funktionsnamens, gefolgt von öffnender runder Klammer und schließender runder Klammer. Zwischen den runden Klammern steht nichts (ggf. Zwischenraumzeichen oder Kommentar!).

Die Bibliotheksfunktion `getchar` ist genau von diesem Typ!

Ist beim Aufruf innerhalb der runden Klammern (versehentlich) ein Argument angegeben, so wird der Compiler eine Fehlermeldung ausgeben!

Die hier definierte Funktion `getint` ist in dieser Form nur eingeschränkt sinnvoll, da das Ende der Eingabe nicht im Funktionsergebnis zurückgegeben werden kann. Die Funktion müsste jeden `int`-Wert (bei korrektem Lesevorgang) und eine Kennung für EOF (bei nicht korrektem Lesevorgang) zurückgeben können. Hierzu ist der Ergebnistyp `int` nicht in der Lage!

6.4 Ergebnistypen

Als Ergebnistyp einer Funktion ist jeder einfache Datentyp möglich! (Daneben gibt es noch einige weitere mögliche Ergebnistypen, welche wir erst später kennenlernen werden!)

Felder (egal von welchem Typ) sind als Funktionsergebnisse **nicht** möglich!

Zu jedem elementaren Datentyp jeder Ausprägung (also `signed char`, `unsigned char`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `float`, `double` und `long double`) kann man also Funktionen definieren, welche einen Wert vom entsprechenden Typ als Funktionsergebnis zurückgibt. Man muss dazu nur jeweils bei Funktionsdeklaration und Funktionsdefinition den gewünschten Typ vor den Namen der Funktion angeben und innerhalb der Funktionsdefinition einen Wert vom gewünschten Typ “erzeugen“ (berechnen oder sich sonstwie verschaffen) und diesen Wert durch eine entsprechende `return`-Anweisung zurückgeben!

Manchmal werden Funktionen benötigt, welche “nur“ etwas erledigen und keinen Funktionswert liefern sollen!

Solche Funktionen sind dadurch gekennzeichnet, dass bei Deklaration und Definition als Ersatz für die Typangabe des Ergebnistypes jeweils das Schlüsselwort `void` angegeben ist.

Solche Funktionen sind bei ihrer Definition durch einfache `return`-Anweisungen:

```
return;
```

ohne zurückzugebenden Wert bzw. durch die den Anweisungsteil der Funktion abschließende geschweifte Klammer ohne vorherige `return`-Anweisung zu beenden!

Der Aufruf solcher Funktionen erfolgt dann (i. Allg.) als einfache Anweisung, d.h. nicht innerhalb komplexerer Ausdrücke (der Funktionsaufruf liefert keinen Wert, der weiterverarbeitet werden könnte!).

In C ist es üblich, auch Funktionen, welche “nur etwas erledigen sollen“ und kein “eigentliches Funktionsergebnis“ haben, trotzdem mit einem Ergebnistyp zu definieren und als “Funktionsergebnis“ einen Fehlerstatus (etwa einen positiven Wert bei fehlerfreiem Ablauf der Funktion und einen negativen bei nicht fehlerfreiem Ablauf der Funktion) zurückzugeben.

So gibt ja auch die Funktion `scanf`, die ja eigentlich nur etwas von der Eingabe lesen und entsprechenden Variablen zuweisen soll, als Funktionsergebnis eine Zahl zurück, an der abgelesen werden kann, ob die Funktion so, wie sie sollte, abgelaufen ist!

Auch die Funktion `printf`, die ja auch nur etwas auf dem Bildschirm ausgeben soll,

hat ein Funktionsergebnis (vom Typ `int`), nämlich die Anzahl der bei diesem Funktionsaufruf ausgegebenen Zeichen bzw. eine negative Zahl im Fehlerfall!

Natürlich kann man auch Funktionen definieren, welche keine Argumente und kein Ergebnis besitzen. Eine solche Funktion wäre wie folgt:

```
void fkt(void);
```

zu deklarieren und entsprechend zu definieren!

6.5 Verwendung mehrerer Funktionen, Rekursion

Man kann zur Lösung eines Problems mehrere Funktionen schreiben und all diese Funktionen in `main` aufrufen.

Es ist auch möglich, innerhalb einer eigenen Funktion (welche etwa von `main` aufgerufen wird) selbst wieder (andere) Funktionen (eigene oder auch Bibliotheksfunktionen) aufzurufen.

Die entsprechende Funktion muss vor ihrem Aufruf nur bekannt (deklariert) sein!

Die Deklaration der Bibliotheksfunktionen erfolgt durch einmaliges Einbinden der entsprechenden Header-Datei, d.h. ist eine Header-Datei einmal eingebunden, so sind die entsprechenden Bibliotheksfunktionen in allen unterhalb der betreffenden `include`-Anweisung definierten Funktionen bekannt und aufrufbar!

Ist eine selbstgeschriebene Funktion `f` innerhalb (zu Beginn) des Anweisungsteils einer Funktion `g` deklariert, so ist sie zunächst nur innerhalb dieses Anweisungsteils von `g` bekannt und aufrufbar! Soll die Funktion `f` auch in einer weiteren Funktion `h` verwendet werden, so ist sie (am Anfang) im Anweisungsteil von `h` erneut zu deklarieren!

Deklarationen von Funktionen können auch außerhalb (extern) erfolgen. Die Deklaration steht dann nicht innerhalb einer Funktionsdefinition am Anfang des Anweisungsteils einer Funktion, sondern etwa vor der Kopfzeile der Funktionsdefinition von `main` oder hinter der schließenden geschweiften Klammer des Anweisungsteils einer Funktion vor der Kopfzeile der nächsten Funktionsdefinition. Eine so deklarierte Funktion ist von der Stelle ihrer Deklaration an bis ans Ende des Quelltextes bekannt und kann in allen dort definierten Funktionen verwendet werden!

Unterhalb der Kopfzeile der Definition einer Funktion braucht diese Funktion nicht mehr deklariert zu werden (weitere Deklarationen stören aber nicht!), d.h. die Kopfzeile einer Funktionsdefinition ist gleichwertig zu einer externen Deklaration dieser Funktion! (Achtung: eine Funktion kann mehrfach deklariert werden, sie darf und muss aber nur genau einmal definiert sein!)

In folgendem Bild markieren die eingerahmten Programmfragmente die Funktionsdefinitionen und die senkrechten Striche rechts davon die Breiche, an denen die entsprechenden Funktionen deklariert und aufrufbar sind:

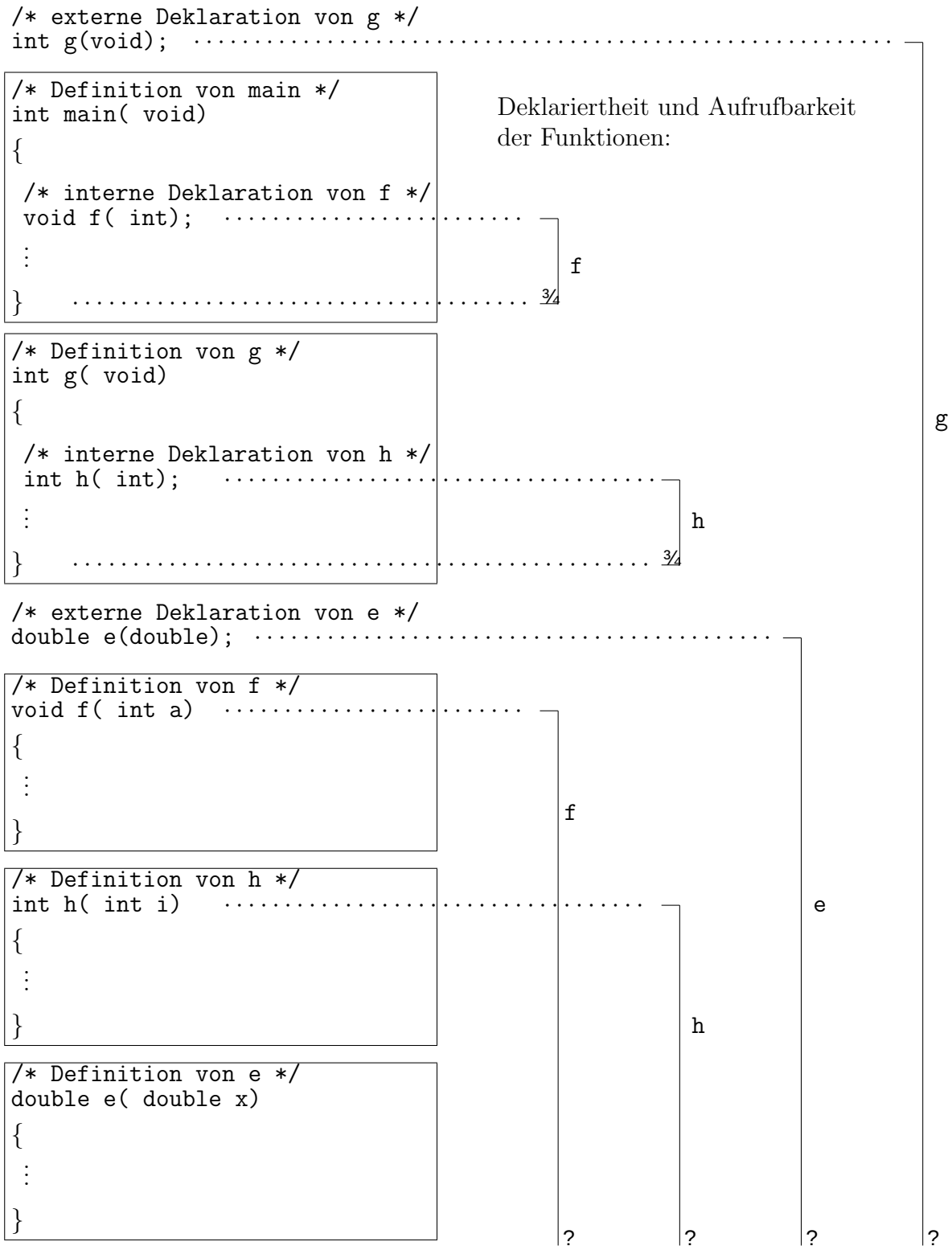


Abbildung 24: Aufrufbarkeit einer Funktion

In der in obigem Bild skizzierten Situation

- kann die Funktion **g** von jeder anderen Funktion aufgerufen werden,
- kann die Funktion **f** nur innerhalb von **main** und innerhalb aller Funktionen, welche unterhalb der Definition von **f** definiert sind, aufgerufen werden,
- kann die Funktion **h** nur innerhalb der Funktion **g** und innerhalb aller Funktionen, welche unterhalb der Definition von **h** definiert sind, aufgerufen werden,
- und die Funktion **e** kann von allen Funktionen, welche unterhalb der Deklaration von **e** definiert sind, aufgerufen werden.

Aus dem Schaubild ist z.B. auch ersichtlich, dass innerhalb der Funktion **f** die Funktion **e** aufgerufen werden kann und dass innerhalb der Funktion **e** wiederum die Funktion **f** aufgerufen werden kann!

Insbesondere ist jede Funktion in ihrem eigenen Anweisungsteil sich selbst bekannt und könnte sich somit selbst wieder aufrufen! (Funktionen, welche sich im Anweisungsteil selbst aufrufen, heissen *rekursive Funktionen*!)

Natürlich muss man Vorkehrungen treffen, dass keine endlosen Aufrufschleifen entstehen:

f ruft **f** und **f** ruft **f** und **f** ruft **f** und **f** ruft **f** und **f** ruft **f** und **f** ruft **f** und **f** ruft **f** und **f** ruft **f**

(Zum einen würde die Abarbeitung theoretisch unendlich lange dauern aber zum anderen wird bei jedem neuen Funktionsaufruf neuer Speicherbereich, etwa für lokale Variablen und Parameter der Funktion, reserviert. Dieser Speicherbereich wird erst beim Funktionsende wieder freigegeben — aber das Ende wird ja nicht erreicht, da vorher die Funktion erneut aufgerufen wird! Da der zur Verfügung stehende Speicherbereich in der Praxis nur eine endliche Größe hat, ist dieser bald aufgebraucht und das Programm endet mit einer System-Fehlermeldung!)

Es gibt durchaus sinnvolle Anwendungen rekursiver Funktionen!

In derartigen Funktionen ist dann immer mindestens ein “Ende-Sonderfall” eingebaut, welcher durch die Funktion direkt und nicht rekursiv erledigt wird.

Zur Bearbeitung eines anderen Falls wird dann die Funktion selber wieder aufgerufen, wobei die Parameter bei diesem erneuten Aufruf dann so sind, dass die Situation in der neu aufgerufenen Reinkarnation der Funktion einen Schritt “näher” am Ende-Sonderfall ist als in der aufrufenden Funktion. So wird nach endlich vielen Schritten und Selbstaufrufen der Funktion schließlich bei einem Aufruf der Ende-Sonderfall erreicht — dieser wird direkt bearbeitet und dieser letzte Funktionsaufruf ist beendet, worauf der Programmfluss in der als vorletzte aufgerufenen Funktion (hinter der Stelle, wo der Aufruf zur letzten erfolgte) weitergeht. Nun kann die vorletzte zu Ende abgearbeitet werden, worauf es dann in der drittletzten weitergeht usw., bis schließlich auch der erste Funktionsaufruf beendet ist!

Als Beispiel wollen wir uns folgende rekursive Version einer Funktion zur Berechnung von Fakultäten ansehen (Fakultäten kann man natürlich auch — und besser — durch eine einfache Schleife berechnen!):

```

long fakul ( int n)
{ long fak;

  /* Ende-Sonderfall : n kleiner gleich 1 */
  if ( n <= 1 ) return 1;

  /* rekursiver Aufruf mit Argument ( n - 1 ),
     also eins naeher am Ende-Sonderfall */
  fak = n * fakul( n - 1 );

  return fak;
}

```

Beim Aufruf von `erg = fakul(4);`, etwa im Hauptprogramm ergibt sich dann folgende Hierarchie von Aufrufen:

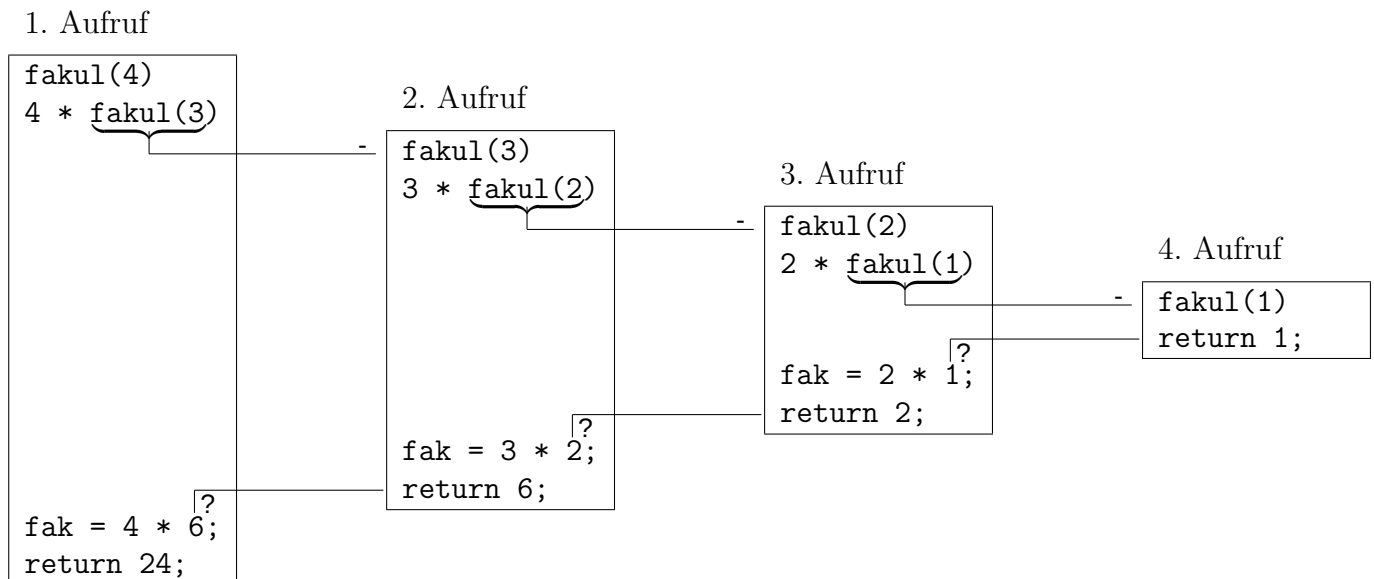


Abbildung 25: Aufruf einer rekursiven Funktion

Durch Rekursion werden Programmläufe i. Allg. nicht schneller, brauchen meistens jedoch mehr Speicherplatz als Lösungen ohne Rekursion!

Hin und wieder lassen sich jedoch schwierige Probleme elegant durch Rekursion lösen!

6.6 Wertverändernde Funktionen, Adress-Variablen

Wir haben bereits erfahren, dass Funktionsparameter lokale Kopien (in der aufgerufenen Funktion) der Funktionsargumente (der aufrufenden Funktion) sind! Wird die Parametervariable in der Funktion abgeändert, so bleibt im aufrufenden Programmteil das korrespondierende Funktionsargument unverändert! Aus diesem Grund schlägt

folgender Versuch fehl, den Inhalt zweier Variablen durch eine Funktion zu vertauschen:

```
void swap( int a, int b)
{ int tmp;
  tmp = a;
  a = b;
  b = tmp;
  return;
}
```

Ruft man nämlich die Funktion wie folgt auf:

```
...
int i = 5;
int j = 7;
...
swap( i, j);
...
```

so hat nach diesem Aufruf *i* immer noch den Wert 5 und *j* den Wert 7! (Beim Aufruf erhält der Funktionsparameter *a* den Wert des korrespondierenden Argumentes *i*, also 5 und der Parameter *b* erhält den Wert von *j*, also 7. In der Funktion werden nun mit Hilfe der temporären Variablen *tmp* die Werte der lokalen Variablen *a* und *b* vertauscht, aber dann ist die Funktion beendet und *i* und *j* im aufrufenden Programmteil sind noch immer so wie vorher!)

Um die Vertauschung der Variablen des aufrufenden Programmteils innerhalb der Funktion durchführen zu können, müsste man innerhalb der Funktion auf die Variablen des aufrufenden Programmteils zugreifen können! Sollen also die Variablen *i* und *j* vertauscht werden, so müsste in der Funktion auf *i* und *j* des aufrufenden Programmteils zugegriffen werden und sollen die Variablen *k* und *l* vertauscht werden, so müsste in der Funktion auf *k* und *l* des aufrufenden Programmteils zugegriffen werden.

Um den Mechanismus zu verstehen, den die Programmiersprache C hierfür zur Verfügung stellt, wollen wir rekapitulieren, welche Merkmale eine Variable besitzt — jede Variable hat:

- Einen Namen.
Mit diesem Namen wird die Variable im Quelltext angesprochen.
- Einen Typ.
Der Typ legt fest, wieviel Speicherplatz im Arbeitsspeicher für diese Variable reserviert wird und welche Operationen mit dieser Variablen möglich sind.
- Eben diesen Speicherbereich im Arbeitsspeicher. (Eine **char**-Variable hat einen Bereich von einem Byte, eine **float**-Variable (etwa) einen von vier Byte!) Für die Reservierung dieses Bereiches sorgt das System!
- Einen Wert.
Die Bit-Kombination im Speicherbereich wird als Wert im betreffenden Typ

interpretiert!

Ein- und dieselbe Bitkombination etwa von der Länge vier Byte steht für unterschiedliche Werte, je nachdem ob diese Bitkombination in einem für eine `int`-Variable reservierten Bereich liegt und entsprechend als `int`-Wert aufgefasst wird oder ob diese Bitkombination in einem für eine `float`-Variable reservierten Bereich vorliegt und entsprechend als `float`-Wert aufgefasst wird!

Unser Augenmerk gilt jetzt diesem für die Variable reservierten Speicherbereich irgendwo im Arbeitsspeicher. Der Arbeitsspeicher kann als Folge von durchnummerierten Bytes aufgefasst werden!

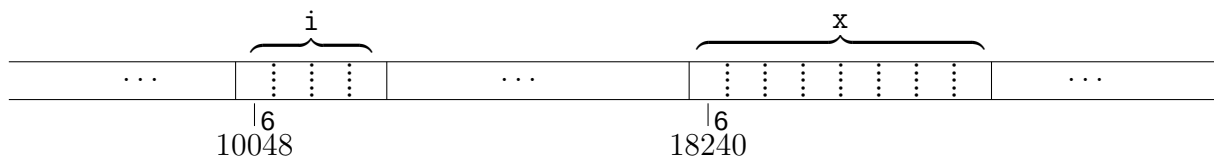
Bei der Definition

```
int i;
```

```
double x;
```

werden dann für `i` (etwa) 4 aufeinanderfolgende Bytes und für `x` (etwa) 8 aufeinanderfolgende Bytes im Arbeitsspeicher reserviert. Jedes dieser reservierten Bytes hat natürlich eine Nummer (die Bytes im Arbeitsspeicher "sind" durchnummeriert!)

Für Variablen ist die Nummer ihres ersten Bytes, ihre sog. *Adresse*, von Bedeutung:



Für den Rechner ist (intern) wichtig, dass an der Stelle 10048 eine `int`-Variable definiert ist (und somit — auf UNIX-Rechnern) dieses und die nächsten drei Byte für die entsprechende Variable reserviert sind und dass an der Stelle 18240 eine `double`-Variable beginnt!

Diese Adressen — die Zahlen selbst sind hier fiktiv — sind natürlich ganzzahlig, sie haben aber jeweils Typen, nämlich

- 10048 ist die Adresse einer `int`-Variablen (an diesem Typ ist ersichtlich, dass dieses und die nächsten drei Byte zu dieser Variablen gehören) und
- 18240 ist die Adresse einer `double`-Variablen (an diesem Typ ist ersichtlich, dass dieses und die nächsten sieben Byte zu dieser `double`-Variablen gehören)!

Somit haben wir ein weiteres Merkmal zu einer Variablen, nämlich ihre Adresse und Adressen haben einen Typen (die Adresse einer `int`-Variablen ist vom Typ etwas anderes als die Adresse einer `double`-Variablen und der ganzzahlige Wert einer Adresse ist etwas anderes als meinetwegen ein `int`-Wert!).

Das einzige Merkmal einer Variablen, welches im Lauf des Programmes vom Programm abgeändert werden kann, ist ihr Wert.

Name, Typ, Speicherbereich und Adresse des Speicherbereiches können vom Programm nicht abgeändert werden!

(Beachte: Speicherbereich und somit Adresse einer Variablen werden bei ihrer Erzeugung vergeben und an ihrem "Lebensende" wird der Speicherbereich wieder freigegeben! Eine lokale Variable einer Funktion wird beim Aufruf der Funktion neu erzeugt und "stirbt" beim Funktionsende! Hier kann nun auftreten, dass ein- und dieselbe lokale Variable der Funktion beim einen Aufruf der Funktion die eine Adresse be-

kommt und bei einem anderen eine andere! Insbesondere lebt bei einem rekursiven Aufruf der Funktion eine lokale Variable noch, wenn die Funktion rekursiv nochmals aufgerufen wird und “dieselbe“ Variable wird bei diesem Aufruf nochmals, an anderer Stelle erzeugt — zu jedem begonnenen und noch nicht abgeschlossenen Aufruf der Funktion existiert somit eine eigene Verwirklichung dieser Variablen mit eigenem Speicherbereich und eigener Adresse!)

6.6.1 Der Adress-Operator

Die Adresse einer Variablen ist i. Allg. für den Programmierer unerheblich und er kann sinnvoll mit dieser Variablen arbeiten, ohne ihre Adresse zu kennen! (Der Rechner muss die Adresse kennen, um auf den entsprechenden Speicherbereich zugreifen zu können!)

Trotzdem gibt es in C den sog. Adress-Operator `&`, welcher zu einer Variablen deren Adresse als Wert (vom entsprechenden Typen) liefert!

Wendet man beispielsweise diesen Operator auf unsere obige Variable `i` an: `&i`, so hat dieser Ausdruck den Wert 10048 und der Typ dieser Zahl ist: Adresse einer `int`-Variablen.

Entsprechend hat der Ausdruck `&x` mit der obigen `double`-Variablen `x` den Wert 18240 und der Typ dieser Zahl ist: Adresse einer `double`-Variablen.

Wie bereits erwähnt braucht ein Programmierer i. Allg. selbst nicht mit Adressen zu arbeiten, das macht das System.

Trotzdem kann er sich beliebige Adresswerte ausgeben lassen. Hierzu kann die `printf`-Umwandlungsspezifikation `%p` verwendet werden, etwa:

```
printf("i: Wert: %d  Adresse: %p\n", i, &i);  
printf("x: Wert: %g  Adresse: %p\n", x, &x);
```

Im Beispielprogramm `funktion/adresse.c` werden zu einigen Variablen des Hauptprogramms und aufgerufener Funktionen deren Adressen ausgegeben!

6.6.2 Adress-Variablen

Neben dem Adress-Operator gibt es in C sog. *Adress-Variablen*, welche als Wert die Adresse von anderen Variablen aufnehmen können!

Genau wie Adressen an einen Typ gebunden sind (etwa Adresse einer `int`-Variablen oder Adresse einer `double`-Variablen) sind auch Adress-Variablen an einen Typen gebunden: z.B. gibt es `int`-Adress-Variablen (welche als Wert nur eine “`int`-Adresse“ — d.h. die Adresse einer `int`-Variablen — aufnehmen können) und `double`-Adress-Variablen (welche als Wert nur eine “`double`-Adresse“ — d.h. die Adresse einer `double`-Variablen — aufnehmen können)!

Adress-Variablen werden wie gewöhnliche Variablen definiert, nur dass vor ihrem Namen ein Stern `*` stehen muss, etwa:

```
int    i, j, *ip;  
double x, y, *dp;
```

Die Variable `ip` ist eine `int`-Adress-Variable, darf als Wert also nur `int`-Adressen aufnehmen und `dp` ist eine `double`-Adress-Variable, darf als Wert also nur `double`-Adressen aufnehmen.

Folgende Zuweisungen sind somit zulässig:

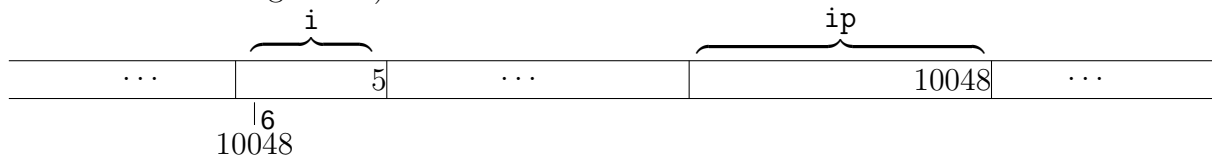
```
ip = &i; /* ip erhaelt als Wert die Adresse von i */
dp = &x; /* dp erhaelt als Wert die Adresse von x */
...
ip = &j; /* ip erhaelt als Wert die Adresse von j */
dp = &y; /* dp erhaelt als Wert die Adresse von y */
```

Die Zuweisung

```
ip = &x; /* Fehler: falscher Typ */
```

ist nicht zulässig und führt zu einer Fehlermeldung!

Habe die Variable `i` den Wert 5 und die Adresse 10048, so ist nach der (korrekten) Zuweisung `ip = &i`; die Speicherbelegung etwa wie folgt (die Zahlen im Speicher sind natürlich binär dargestellt!):



Ansonsten sind Adress-Variablen gewöhnliche Variablen, haben einen Namen (etwa `ip`), einen Typen (etwa: `int`-Adress-Variable), einen Speicherbereich (auch Adress-Variablen werden in einem Speicherbereich des Arbeitsspeichers abgelegt — und dieser hat wiederum selbst eine Adresse!) und einen Wert (etwa kann `ip` als Wert die Adresse der `int`-Variablen `i` haben!).

Nach ihrer Definition hat eine Adress-Variable, genau wie jede andere Variable, i. Allg. einen zufälligen Wert, es sei denn, sie wird bei ihrer Definition explizit initialisiert — was natürlich auch bei Adress-Variablen möglich ist, etwa:

```
int i, *ip = &i;
```

Außer der Adresse einer entsprechenden Variablen kann einer Adress-Variablen der Wert `NULL` (ebenfalls eine vom Standard definierte symbolische Konstante mit dem Wert 0) zugewiesen werden. Hat eine Adress-Variable diesen Wert `NULL`, so bedeutet das, dass sie im Augenblick nicht mit einer gültigen Adresse (einer Variablen vom entsprechenden Typ) belegt ist.

6.6.3 Der Verweis-Operator

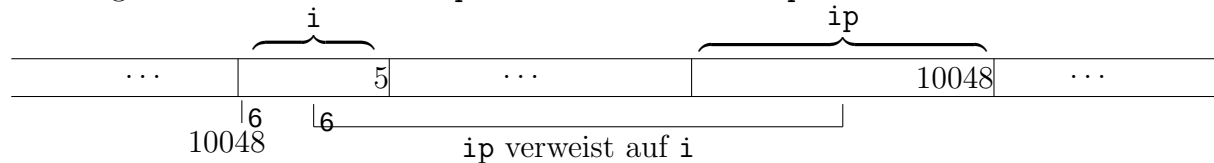
Was kann man nun mit Adressen und Adress-Variablen anfangen?

In Umkehrung zum Adress-Operator `&`, der zu einer Variablen deren Adresse liefert, gibt es den *Verweis-Operator* `*`, der zu einer Adresse die Variable liefert, welche an der entsprechenden Adresse abgespeichert ist!

Ist `ip` eine `int`-Adress-Variable, so ist `*ip` die `int`-Variable, deren Adresse in `ip` abgespeichert ist! Nach der Zuweisung `ip = &i`; mit einer `int`-Variablen `i` kann im

Programm mittels `*ip` auf diese Variable `i` zugegriffen werden!

Man sagt: die Adress-Variable `ip` verweist auf `i` bzw. `*ip` "ist" `i`.



Im nachfolgenden Programmtext kann `*ip` jetzt überall da verwendet werden, wo auch die `int`-Variable, deren Adresse in `ip` steht, verwendet werden könnte:

```
...
ip = &i;           /* ip verweist auf i      */
*ip = 8;           /* i wird zu 8                             */
*ip = *ip + 7;     /* i wird um 7 erhoeht                     */
printf("%d\n", *ip); /* i wird ausgegeben                       */
ip = &j;           /* ip verweist auf j                       */
*ip = 10;          /* j wird zu 10, i bleibt unveraendert */
*ip = *ip - 5;     /* j wird um 5 erniedrigt, i bleibt       */
printf("%d\n", *ip); /* j wird ausgegeben                       */
```

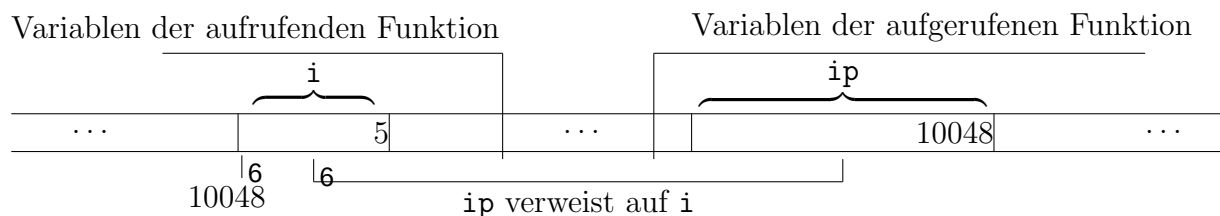
Man kann natürlich gleich mehrere Adress-Variablen auf ein- und dieselbe Variable verweisen lassen:

```
...
int i, *ip1, *ip2; /* i gew. int Variable, */
...               /* ip1 und ip2 int-Adress-Variablen */
ip1 = &i;
ip2 = &i;          /* beide verweisen auf i */
...
*ip1 = *ip2 + 2; /* i wird um 2 erhoeht */
...
```

6.6.4 Wertverändernde Funktionen

Der Zugriff auf eine Variable (des Hauptprogramms) über ihre Adresse funktioniert auch von einer aufgerufenen Funktion aus, wenn die Adresse der Hauptprogrammvariablen (als Wert) in der Funktion bekannt ist.

Steht also in einer Adress-Variable `ip` einer Funktion die Adresse einer Variablen `i` des Hauptprogramms, so kann innerhalb der aufgerufenen Funktion mittels `*ip` auf die Variable `i` des Hauptprogramms zugegriffen werden und etwa durch `*ip = *ip + 7;` ihr Wert verändert werden!



Will man also gezielt aus einer Funktion heraus eine Variable des aufrufenden Programmteils verändern, so muss in der Funktion die Adresse dieser Variablen des aufrufenden Programmteils bekannt sein — man muss also dafür sorgen, dass beim Funktionsaufruf die fragliche Adresse (als Wert) von der aufrufenden Funktion an die aufgerufene Funktion übergeben wird!

Zur Übergabe von Werten bei Funktionsaufrufen dienen bekanntlich die Funktionsparameter, welche beim Aufruf den Wert des zugehörigen Funktionsargumentes erhalten. Soll nun eine Adresse als Wert übergeben werden, so muss der zugehörige Funktionsparameter eine Adress-Variable sein und beim Aufruf ist als Funktionsargument die Adresse der durch die Funktion zu ändernden Variablen anzugeben!

Insbesondere dürfen Funktionsparameter vom Adresstyp sein, d.h. man kann Funktionen definieren, deren Funktionsparameter `int`-Adress-Variablen oder `double`-Adress-Variablen usw. sind!

Sogar als Ergebnistyp von Funktionen sind Adresstypen zugelassen, d.h. man kann Funktionen definieren, welche als Funktionswert beispielsweise die Adresse einer `int`-Variablen liefern!

Zurück zu unserer (missglückten) Vertauschung der Werte zweier `int`-Variablen durch eine Funktion!

Sollen die Inhalte zweier `int`-Variablen durch eine Funktion vertauscht werden (insbesondere soll jede der Variablen dabei verändert werden!), so müssen der Funktion die Adressen der zu vertauschenden Variablen übergeben werden — die Funktionsparameter müssen entsprechend Adress-Variablen sein und die Funktion ist wie folgt zu deklarieren:

```
void swap (int *, int *);
```

Diese Funktion mit dem Namen `swap` hat also keinen Funktionswert (`void` vor dem Namen) und benötigt zwei `int`-Adressen als Funktionsargumente. (`int *` ist als: Adresse einer `int`-Variablen zu lesen!)

Die Definition der Funktion könnte so aussehen:

```
void swap( int *a, int *b)
{ int tmp;
  tmp = *a;
  *a = *b;
  *b = tmp;
  return;
}
```

Die Funktionsparameter sind also zwei `int`-Adress-Variablen mit den Namen `a` bzw. `b`.

Sind im Hauptprogramm zwei `int`-Variablen `i` und `j` definiert, deren Werte durch diese Funktion vertauscht werden sollen, so ist der korrekte Aufruf der Funktion wie folgt:

```
swap( &i, &j);
```

Die Funktion benötigt als Argument zwei `int`-Adressen und `&i` und `&j` sind genau vom richtigen Typ. Diese beiden Adressen werden in die lokalen Adress-Variablen `a`

und `b` abgespeichert und in der Funktion sind somit die Adresse von `i` (steht in `a`) und die Adresse von `j` (steht in `b`) bekannt. (`i` und `j` sind Variablen des Hauptprogramms, aber deren Adressen stehen nach dem Aufruf der Funktion in den lokalen Parametervariablen `a` und `b` der Funktion!)

Im Ablauf der Funktion wird mit `*a` der Speicherbereich der (Hauptprogramm-)Variablen `i` und mit `*b` der Speicherbereich der (Hauptprogramm-)Variablen `j` angesprochen und auf diese Variablen kann von der Funktion aus zugegriffen — hier mit der üblichen Technik unter Verwendung einer lokalen Hilfsvariablen deren Inhalte vertauscht — werden.

Will man also in einer Funktion den Wert einer Variablen (etwa `i`) des aufrufenden Programnteils verändern, so ist nicht die Variable selbst (`i`) als Argument zu übergeben, sondern deren Adresse (`&i`) und die Funktion muss man entsprechend definieren und deklarieren: der entsprechende Parameter muss vom Adresstyp sein (hier: `int *`).

Dies ist auch der Grund für die Asymmetrie zwischen den Bibliotheksfunktionen `printf` und `scanf` bei der Ausgabe von Variablen bzw. beim Einlesen von Variablen! Bei der Ausgabe wird der Wert einer Variablen nicht verändert und die Variable selbst (ihr Name) ist als Funktionsargument anzugeben:

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

Bei der Eingabe soll die Variable ja gerade einen neuen Wert bekommen und entsprechend muss beim Aufruf nicht der Name der Variablen, sondern die Adresse der Variablen als Funktionsargument angegeben werden:

```
scanf("%d", &i);
```

6.6.5 Typlose Adressen

Neben den oben beschriebenen typgebundenen Adressen und Adress-Variablen gibt es in C auch typlose Adressen und entsprechende Adress-Variablen.

Typlose Adressen sind einfach “Speicheradressen” (Nummer eines Bytes im Arbeitsspeicher), ohne dass von Interesse ist, was sich an dieser Stelle im Arbeitsspeicher befindet!

Adress-Variablen von diesem “typlosen Typ” werden wie folgt definiert:

```
void *p;
```

(`void` ist zwar kein C-Datentyp, aber `void *` ist einer!)

Diese Adress-Variable `p` kann nun eine beliebige Speicheradresse aufnehmen, die Zuweisungen `p = &i;` und `p = &x;` mit einer `int`-Variablen `i` und einer `double`-Variablen `x` sind zulässig, wobei jedoch an der in `p` abgespeicherten Adresse nicht mehr der Typ des Objektes erkennen lässt, welches an dieser Stelle im Arbeitsspeicher abgelegt ist!

Da bei einer typlosen Adresse der Typ des Objektes, welches an dieser Adresse steht, nicht bekannt ist, kann und darf der Verweis-Operator `*` nicht auf solche typlosen Adressen oder `void`-Adress-Variablen angewendet werden!

6.6.6 Felder und Adressen von Adress-Variablen

Natürlich kann man auch Felder von Adress-Variablen definieren, etwa:

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

`a` ist ein Feld der Länge 10 und jedes Feldelement ist eine `int`-Adress-Variable (Typ: `int *`). D.h. `a[0]` bis `a[9]` sind `int`-Adress-Variablen und mit diesen kann alles gemacht werden, was mit sonstigen `int`-Adress-Variablen gemacht werden kann!

Adress-Variablen haben ansonsten genau die gleichen Merkmale wie andere Variablen auch: sie haben einen Namen, einen Typen, einen Wert und einen zugehörigen Speicherbereich. Dieser Speicherbereich hat natürlich auch eine Adresse. Die Adresse einer Adress-Variable kann wiederum in einer entsprechenden Adress-Variable abgelegt werden.

Beispiel:

```
int i, *ip, **ipp;
```

Die Variable `i` ist eine `int`-Variable.

Die Variable `ip` ist eine Adress-Variable vom Typ `int`, d.h. sie darf als Wert die Adresse einer `int`-Variablen aufnehmen — die Zuweisung `ip = &i;` ist also korrekt.

Die Variable `ipp` ist eine Adress-Variable vom Typ `int *`, d.h. sie darf als Wert die Adresse einer `int *`-Variablen aufnehmen — die Zuweisung `ipp = &ip;` ist also korrekt, nicht jedoch die Zuweisung `ipp = &i`, da `&i` die Adresse einer `int`-Variablen und eben nicht die Adresse einer `int`-Adress-Variable ist.

Die Hierarchie von Adress-Variablen lässt sich weiter fortsetzen — hier soll aber nicht weiter darauf eingegangen werden.

Eine der wenigen Situationen, wo bei “normalen” Programmen Adressen von Adress-Variablen auftauchen ist die, wenn durch eine Funktion der Wert einer Adress-Variablen verändert werden soll. Wegen des Prinzips: *Call by Value* muss dieser Funktion nicht die Adress-Variable selbst, sondern die Adresse der zu abzuändernden Adress-Variablen übergeben werden.

Soll also eine etwa durch `int *ip;` definierte Variable durch eine Funktion abgeändert werden, so müsste diese Funktion mit folgender Kopfzeile definiert bzw. definiert werden:

```
... fkt( int **,...)
```

und der Aufruf müsste wie folgt geschehen:

```
... fkt( &ip, ...) ...;
```

6.7 Felder und Funktionen

6.7.1 Eindimensionale Felder als Funktionsargumente

Man kann Funktionsparameter vom Feldtyp definieren, etwa:

Definition:

```
int f( int a[100], double x)
```

```
{ ... }
```

Bei der Deklaration reicht als Typangabe `int []`, also Feld vom Typ `int` (Name und Feldlänge sind hier nicht erforderlich!):

```
int f( int [], double);
```

Auch bei der Definition der Funktion braucht die Feldlänge nicht angegeben zu werden, sie wird ohnehin vom Compiler ignoriert!

Dieser setzt sowohl bei Deklaration und Definition den Typ: `int []`, also Feld vom Typ `int` um in den Typen: `int *`, also Adress-Variable vom Typ `int`.

Ein Feld eines gewissen Types als Funktionsparameter wird vom Compiler also genauso behandelt wie eine Adress-Variable vom entsprechenden Typ!

Für unseren Funktionsparameter `int a[100]` in obiger Funktionsdefinition wird also durch den Compiler nur eine Adress-Variable vom Typ `int` als lokale Parametervariable definiert!

Was geschieht nun bei dem Funktionsaufruf:

```
...
int feld[50], i;
double x;
...
i = f ( feld, x);
...
```

Der erste Funktionsparameter ist eine Adress-Variable vom Typ `int`, das erste Argument aber der Name eines Feldes vom Typ `int`!

Hier muss folgende C-Regel beachtet werden:

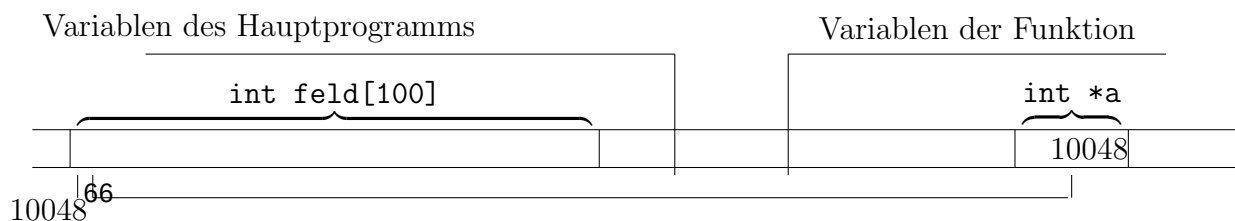
Der Name eines Feldes hat einen Wert und einen Typen.

Der Wert ist die Adresse des Feldanfangs und

der Typ ist Adresse des dem Feld zugrundeliegenden Types.

Bei obigem Funktionsaufruf steht als erstes Argument der Name `feld`. Dieser Name hat einen Wert: Nummer des ersten Bytes des für dieses Feld reservierten Speicherbereiches. Dieser Wert hat einen Typen: Adresse einer `int`-Variablen.

Diese Adresse wird jetzt beim Aufruf in die lokale `int`-Adress-Variable `a` der Funktion abgespeichert, d.h. in dieser Parametervariablen vom Typ `int *` steht die Adresse des Feldanfangs des Feldes `feld` aus dem aufrufenden Programmteil!



Insbesondere ist es für die Funktion unerheblich (und in der Funktion nicht bekannt), ob das beim Aufruf angegebene Feld die Länge 50 oder 100 oder 100000 hat. Das einzige, was in der Funktion bekannt ist, ist die Adresse des Feldanfangs!

Aufgrund des Typen `int *` des ersten Funktionsparameters hätte diese Funktion auch völlig gleichwertig mittels `i = f ( &feld[0], x );` aufgerufen werden können. Innerhalb der Funktion kann kein Unterschied zum Aufruf `i = f ( feld, x );` festgestellt werden. In beiden Fällen hat die Parametervariable `a` in der Funktion denselben Wert!

Mittels dieser lokalen `int`-Adress-Variablen `a` könnte — wie üblich — innerhalb der Funktion mittels des Verweis-Operators auf die `int`-Variable (des Hauptprogramms), deren Adresse in `a` abgespeichert ist (also auf die Variable `feld[0]`) zugegriffen und diese auch aus der Funktion heraus verändert werden:

```
*a = 23;
```

und das Feldelement `feld[0]` des Feldes des Hauptprogramms erhält den neuen Wert 23.

Alternativ zu dieser Zugriffsart mit dem Verweis-Operator kann mittels Indizierung `a[0]` ebenfalls aus der Funktion heraus auf dieses Element `feld[0]` zugegriffen werden!

Hier sind auch größere Indizes möglich: `a[1]` bezieht sich auf `feld[1]`, `a[2]` bezieht sich auf `feld[2]` usw.!

Hier tritt eine weitere Regel zu Tage:

Eine Adresse darf wie der Name eines Feldes verwendet werden! Diese Adresse wird dann als die Anfangsadresse eines Feldes angesehen und das System geht davon aus, dass an dieser Adresse tatsächlich ein Feld des zugrundeliegenden Types beginnt! Mit Indizierung wird dann auf die entsprechenden Feldelemente zugegriffen!

Verwendet man eine Adress-Variable wie einen Feldnamen (d.h. hinter dem Namen der Adress-Variablen steht in eckigen Klammern ein Index), so muss man selbst dafür sorgen, dass an der Adresse, welche in der Adress-Variablen abgespeichert ist, tatsächlich ein Feld beginnt!

Dies ist aber bei Funktionsparametern vom Adress-Typ und Feldern als Funktionsargumenten gewährleistet!

Aus dem Geschilderten ist ersichtlich, dass bei einem Feld als Funktionsargument dessen Anfangsadresse in eine lokale Adress-Variable der Funktion “hineinkopiert” wird. D.h. trotz des Prinzips: *Call by Value* werden Felder nicht als Ganzes in lokale Felder kopiert, sondern eben nur ihre Anfangsadresse in eine lokale Adress-Variable. Durch indizierten Zugriff kann somit innerhalb der Funktion das originale Feld des aufrufenden Programmteils abgeändert werden!

Aus diesem Grunde können Felder durch Funktionen abgeändert, den Feldelementen also auch neue Werte zugewiesen werden!

Dies wird im Beispiel `funktion/fibonac3.c` ausgenutzt, um durch eine Funktion die ersten Elemente der Fibonacci-Folge zu berechnen und in einem Feld als Funktionsargument abzuspeichern!

Da bei einem Feld als Funktionsargument nur dessen Anfangsadresse der Funktion mitgeteilt wird, nicht aber (automatisch) dessen Feldlänge, ist es üblich, um innerhalb der Funktion Feldüberschreitungen zu vermeiden, die Feldlänge (oder eine kleinere Zahl, wenn nur ein Teil des Feldes verwendet werden soll) als gewöhnlichen Parameter vom Typ etwa `int` zusätzlich zu übergeben, etwa:

```
int f ( int feld[], int laenge) { ... }
```

und innerhalb der Funktion wird anhand des zusätzlichen Parameters `laenge` durch geeignete Abfragen ein Überschreiten der Feldlänge vermieden! Insbesondere sollte die Überprüfung der Feldindizes **nicht** durch eine gemeinsame symbolische Konstante für Hauptprogramm und Funktion erfolgen, da dann die Funktion nur mit diesem Hauptprogramm zusammen und nur für Felder einer gewissen Länge (und nicht universell) funktioniert!

Diese in natürlicher Weise gegebene Abänderbarkeit von Feldern als Funktionsargumenten innerhalb der Funktion ist auch der Grund dafür, dass zum Einlesen von Zeichenketten mit `scanf` **kein** `&` vor dem Feldnamen zu stehen braucht! (Dies wäre sogar falsch, da der Feldname bereits eine konstante Adresse als Wert hat und der Adress-Operator nicht auf eine Adress-Konstante angewendet werden darf!)

Wie bereits erwähnt, wird eine String-Konstante (konstante Zeichenkette, etwa wie `"hello, world\n"`) intern auch als Feld von (nicht variablen) Zeichen abgespeichert (mit anschließendem `'\0'`).

Auch ein konstantes Zeichenfeld hat einen Wert, nämlich die Adresse des ersten Zeichens des Feldes, und dieser Wert hat den entsprechenden Typen: Adresse eines Zeichens.

Aus diesem Grund könnte eine wie folgt definierte (und entsprechend deklarierte) Funktion:

```
void fkt ( char *a)
{...}
```

mit (formal) unterschiedlichen Argumenten aufgerufen werden:

```
char wort[32], *p;
...
f(p);      /* Argument ist die char-Adress-Variable p */
...       /* Wert von p wird in die lokale Adress-Variable a kopiert */
...
f(wort);   /* Argument ist der Name eines char-Feldes */
...       /* Adresse des Feldanfangs von wort wird in die */
...       /* lokale Adress-Variable a kopiert */
...
f("hello, world\n"); /* Argument ist eine konstante Zeichenkette */
...       /* Adresse des Feldanfangs des konstanten Feldes wird */
...       /* in die lokale Adress-Variable a kopiert */
```

Ein Unterschied ist hierbei ggf. zur Laufzeit des Programms zu bemerken: wird innerhalb dieser Funktion `f` etwa versucht, den Wert von `a[0]` abzuändern, so ist das bei den ersten beiden Aufrufformen möglich, bei der letzten Aufrufform jedoch nicht, da in diesem Fall `a[0]` nichts anderes als das erste (konstante) Zeichen `'h'` der konstanten Zeichenkette `"hello, world\n"` und das Abändern von Konstanten durch das System verhindert wird (bzw. werden sollte!).

Um deutlich zu machen, dass eine derartige Funktion sowohl für variable, als auch für konstante Zeichenketten aufrufbar ist und demnach die Zeichenkette durch die Funktion nicht verändert wird, sollte die Funktion wie folgt definiert und entsprechend deklariert werden, vgl. Abschnitt 7.4.2:

```
void f( const char * a) { ... }
```

Obwohl Adress-Variablen und Feldnamen in C-Programmen weitgehend gleich verwendet werden können (auf Adress-Variablen kann der Feldindizierungsoperator `[]` und auf Feldnamen kann der Verweis-Operator `*` angewendet werden und beide können einer Funktion als Argument übergeben werden, welche eine entsprechende Adress-Variable als Parameter hat!) bestehen doch signifikante Unterschiede:

- Bei einem Feld ist automatisch der zugehörige Speicherbereich reserviert. Verwendet man eine Adress-Variable wie einen Feldnamen, so muss man selber dafür sorgen, dass die Adress-Variable auf einen ausreichend großen Speicherbereich zeigt — dies ist nicht automatisch der Fall.
- Einer Adress-Variablen kann ein neuer Wert zugewiesen werden — einem Feldnamen jedoch nicht.

6.7.2 Funktionsergebnisse vom Adresstyp

Felder sind als Funktionsergebnisse nicht möglich!

Dennoch kann hinter dem `return` einer entsprechend definierten Funktion ein Feldname angegeben werden — in diesem Fall ist wiederum der Wert des Feldnamens die Anfangsadresse des Feldes und somit wird die Anfangsadresse des Feldes als Funktionsergebnis zurückgegeben.

Die Funktion ist natürlich entsprechend zu deklarieren und definieren — der Ergebnistyp muss ein Adresstyp sein, etwa:

```
int * fkt1(...);
```

für eine Funktion, welche eine `int`-Adresse, oder

```
double *fkt2(...);
```

für eine Funktion, welche eine `double`-Adresse, oder

```
void * fkt3(...);
```

für eine Funktion, welche eine typlose Adresse als Funktionsergebnis liefern soll.

Bei der Rückgabe von Adressen ist zu beachten, dass das Objekt, dessen Adresse zurückgegeben wird, nach Beendigung der Funktion noch vorhanden sein muss! So dürfte folgende syntaktisch korrekte Funktion Laufzeitfehler verursachen:

```
/* FEHLER: Rueckgabe der Adresse eines lokalen Objektes */
```

```
int * fkt( ... )
```

```
{ int a;
```

```
...
```

```
return &a;
```

```
}
```

Nach folgendem Aufruf dieser Funktion:

```
p = fkt(...);
```

wobei `p` eine `int`-Adress-Variable sei, ist der Wert dieser Adress-Variablen `p` die Adresse, welche die lokale Variablen `a` der Funktion `fkt` beim Ablauf der Funktion hatte. Diese Variable `a` gibt es nach Ablauf der Funktion aber gar nicht mehr, der zugehörige Speicherbereich wurde beim Funktionsende wieder freigegeben und wird jetzt ggf. vom System für andere Zwecke (andere Variablen von möglicherweise anderem Typ oder für ein anderes, gleichzeitig ablaufendes Programm) verwendet. Der Zugriff mittels `*p` auf diesen anderweitig verwendeten Speicherbereich dürfte unvorhersehbare Konsequenzen haben!

Folgendes Beispiel ist jedoch korrekt und sinnvoll:

Die hier definierte Funktion erhält zwei Zeichenketten als Argument, bestimmt die längere der beiden Zeichenketten und gibt die Anfangsadresse der längeren als Funktionsergebnis zurück:

```
char *longest( char *a, char *b)
{ int i;

  for ( i = 0; (a[i] != '\0') && (b[i] != '\0') ; i = i + 1)
    ; /* zeichenweise durchlaufen, bis ein String zu Ende ist */

  if ( a[i] == '\0' ) return a;
  else return b;
}
```

Wird diese Funktion korrekt mit zwei gültigen String-Adressen als Argumente aufgerufen, so ist das Funktionsergebnis auch eine gültige String-Adresse — nämlich die Adresse der längeren der als Argumente übergebenen Strings!

6.7.3 Mehrdimensionale Felder als Funktionsargumente

Mehrdimensionale Felder als Funktionsargumente werden ebenfalls nicht in lokale (mehrdimensionale) Felder der Funktion kopiert, sondern hier wird der Funktion über interne Adress-Mechanismen das originale Feld des aufrufenden Programnteils zur Verfügung gestellt!

Hier muss man zu korrekten Verwendung des Feldes sowohl bei der Deklaration als auch bei der Definition der Funktion die zweite (ggf. auch die dritte und alle folgenden) Dimensionierungen angeben!

Soll also das Feld:

```
int a[5][10];
```

an eine Funktion `f` übergeben werden, so ist diese wie folgt zu deklarieren und definieren:

Deklaration:

```
... f ( int [][][10], /* weitere Parametertypen */);
```

Definition:

```
... f ( int c[][][10], /* weitere Parameter */) { ... }
```

Hierbei muss jeweils pro Index ein Paar eckiger Klammern aufgeführt sein, im ersten Paar kann, muss nicht, die Dimensionierung angegeben werden (diese wird aber wiederum vom Compiler ignoriert!), in jedem folgenden Paar eckiger Klammern muss die Dimensionierung angegeben werden!

Alternativ könnte die Funktion auch wie folgt deklariert und definiert werden:

Deklaration:

```
... f ( int (*)[10], /* weitere Argumenttypen */);
```

Definition:

```
... f ( int (*c)[10], /* weitere Parameter */) { ... }
```

wobei die runden Klammern um den Stern `*` bzw. um `*c` nicht weggelassen werden dürfen!

(Grund hierfür:

Die Definition `int *c[10];` definiert ein Feld `c` der Länge 10 von `int`-Adress-Variablen, d.h. `c[0]` bis `c[9]` sind jeweils `int`-Adress-Variablen.

Die Definition `int (*c)[10];` definiert eine Adress-Variable `c`, welche als Wert nur Adressen von Feldern der Länge 10 vom Typ `int` aufnehmen darf!)

Die Tatsache, dass man bei mehrdimensionalen Feldern als Funktionsargumente ab der zweiten alle weiteren Dimensionierungen bei Deklaration und Definition der Funktion (als Konstante) angeben muss, machen solche Funktionen äußerst unflexibel:

Eine Funktion, welche für 5×10 -Matrizen reeller Zahlen aufrufbar sein soll, müsste durch:

```
...f (double [][][10],...);
```

deklariert und entsprechend definiert werden. Sie funktioniert aber beispielsweise nicht für 5×5 Matrizen. Hierfür müsste man eine eigene Funktion schreiben (deklarieren und definieren!), welche dann aber nicht für beispielsweise 5×15 -Matrizen funktioniert!

In `funktion/fkt_md.c` wird ein (derart unflexibles) Beispiel für ein mehrdimensionales Feld als Funktionsargument demonstriert!

6.7.4 Alternative zu mehrdimensionalen Feldern als Funktionsargumente

Die im letzten Unterabschnitt beschriebene Unflexibilität von mehrdimensionalen Feldern als Funktionsargumenten kann wie folgt umgangen werden:

Soll ein zweidimensionales Feld (etwa vom Typ `int`) an eine Funktion übergeben werden, so definiere man neben diesem zweidimensionalen Feld zusätzlich ein eindimensionales Feld von entsprechenden Adress-Variablen, wobei die Länge dieses eindimensionalen Feldes von Adress-Variablen der Anzahl der Zeilen des zweidimensionalen Feldes entspricht:

```
int feld[M][N];
```

```
int *p_feld[M];
```

Sodann Sorge man dafür, dass die i -te Adress-Variable des Adress-Feldes `p_feld` auf die i -te Zeile des zweidimensionalen Feldes `feld` "zeigt":

```
for ( i = 0; i < M; i = i + 1)
    p_feld[i] = &feld[i][0];
```

(anstelle von `&feld[i][0]` kann man auch `feld[i]` schreiben — `feld` ist ein Feld von `int`-Feldern, `feld[i]` ist der Name des i -ten dieser Felder und der Name eines Feldes ist die Adresse des Feldanfangs!)

Ist eine Funktion dann mit folgender Kopfzeile definiert bzw. deklariert:

```
... fkt( int *p[], int m, int n, ...)
```

so kann diese Funktion wie folgt aufgerufen werden:

```
... fkt( p_feld, M, N, ...) ...;
```

und zwar unabhängig von der Länge `N` der Zeilen und der Anzahl der Zeilen des zweidimensionalen Matrix `feld`.

Innerhalb dieser Funktion kann die Adress-Variable `p` wie der Name eines zweidimensionalen Feldes verwendet werden, d.h. Zugriffe der Art `p[i][j]` sind möglich.

Tatsächlich ist `p` eine Variable vom Typ `int **` und in `p` ist die Anfangsadresse des Feldes `p_feld` abgespeichert. `p_feld` ist aber ein Feld von `int`-Adress-Variablen, `p[i]` ist somit die i -te Adress-Variable dieses Feldes. Diese Adress-Variable `p[i]` (entspricht `p_feld[i]`) zeigt aber auf die i -te Zeile des zweidimensionalen Feldes `feld` und `p[i][j]` ist somit das j -te Element der i -ten Zeile dieses Feldes!

Die so definierte Funktion kann über diesen Trick universell für beliebige zweidimensionale Felder vom Typ `int` aufgerufen werden — zur Vermeidung von Feldüberschreitungen werden dieser Funktion nur die Feldgrenzen des zweidimensionalen Feldes (als gewöhnliche Funktionsargumente) übergeben werden. (Innerhalb der Funktion können diese Parameter in Schleifen o.ä. als Grenzen verwendet werden!)

Diese Funktion kann somit etwa für eine 5×5 -Matrix, für eine 5×10 -Matrix oder auch für eine 100×1000 -Matrix aufgerufen werden.

Ist das zweidimensionale Feld etwa mittels

```
int feld[50][100];
```

definiert und zeigen die Elemente des Adressfeldes `int p_feld[50]`; wie oben beschrieben auf die einzelnen Zeilen von `feld`, so ist auch folgender Aufruf möglich:

```
... fkt( p_feld, 10, 20, ...) ...;
```

d.h. von der definierten 50×100 -Matrix wird nur die 10×20 -Teilmatrix oben links in der Funktion verwendet! (Man kann so also Teile von zweidimensionalen Feldern übergeben!)

Entsprechende Techniken sind bei höher-dimensionalen Feldern möglich.

6.8 Mehrere Quelltexte

Besteht ein komplexeres Programm aus mehreren selbstgeschriebenen Funktionen, so brauchen die Funktionsdefinitionen nicht alle in ein- und demselben Quelltext zu

stehen.

D.h. man kann im ersten Quelltext, etwa mit dem Namen `teil1.c` eine Funktion (oder auch mehrerer Funktionen) definieren, in einem weiteren Quelltext, etwa mit dem Namen `teil2.c`, weitere zum Programm gehörende Funktionen und in einem dritten Quelltext (etwa mit dem Namen `teil3.c`) die restlichen Funktionen. (Die Definition einer Funktion kann nicht auf mehrere Quelltexte verteilt werden: Eine in einem Quelltext begonnene Funktionsdefinition muss in diesem Quelltext beendet werden!)

Beim Erstellen der Quelltexte ist darauf zu achten, dass jede in einem Quelltext verwendete Funktion vor ihrer Verwendung in diesem Quelltext ordnungsgemäß deklariert wird.

Wird etwa eine Funktion `f`, welche im dritten Quelltext definiert wird, im ersten Quelltext verwendet (aufgerufen), so ist sie in diesem Quelltext zu deklarieren! Wird sie ebenfalls im zweiten Quelltext verwendet, so ist sie dort ebenfalls zu deklarieren. Dies betrifft auch die Funktionen der Standardbibliothek. Sollte etwa irgendwo in der Datei `teil1.c` und in der Datei `teil2.c`, nicht jedoch in Quelltext `teil3`, die Bibliotheksfunktion `printf` (oder eine sonstige Ein-/Ausgabefunktion) aufgerufen werden, so muss in beiden Quelltexten `teil1.c` und `teil2.c` die Include-Datei `stdio.h` eingebunden werden. In `teil3.c` brauch sie nicht (kann aber) eingebunden zu werden

6.8.1 Compilieren von aus mehreren Quelltexten bestehenden Programmen

Auf UNIX-Systemen kann ein aus mehreren Quelltexten (etwa `teil1.c`, `teil2.c` und `teil3.c`) bestehendes Programm wie folgt zu einer lauffähigen Version `a.out` compiliert werden (im Folgenden wird der `gcc`-Compiler verwendet — es könnte völlig gleichwertig mit gleichen Optionen auch der `cc`-Compiler verwendet werden!):

```
gcc teil1.c teil2.c teil3.c<cr>
```

Soll das fertige Programm direkt einen anderen Namen erhalten, so kann wiederum die Option `-o name` verwendet werden.

Man kann die Dateien auch einzeln übersetzen, wobei dem Compiler jeweils (durch die Option `-c`) mitgeteilt werden muss, dass er kein lauffähiges Programm, sondern nur eine *Objektdatei* erstellen soll:

```
gcc -c teil1.c<cr>
```

Hierdurch übersetzt der Compiler den Quelltext `teil1.c` und erstellt hieraus die (übersetzte, aber allein nicht lauffähige) Objektdatei `teil1.o`.

Entsprechend können die anderen Quelltexte einzeln, oder auch zusammen übersetzt werden:

```
gcc -c teil2.c teil3.c<cr>
```

Hiernach sollten zwei weitere Objektdateien `teil2.o` und `teil3.o` entstanden sein. Die Objektdateien müssen jetzt noch (eigentlich durch einen Linker-Aufruf) zusammengebunden werden. Dies geschieht (auch) durch folgenden Aufruf des Compilers:

```
gcc teil1.o teil2.o teil3.o<cr>
```

wobei nicht mehr übersetzt wird, sondern nur noch zum lauffähigen Programm `a.out` zusammengebunden wird. (Auch hier kann die Option `-o name` verwendet werden!) Der Compiler kann somit nicht nur Quelltexte weiterverarbeiten (übersetzen), sondern auch Objektdateien (zusammenbinden).

Beim Compileraufruf können auch Quelltexte und Objektdateien gemischt angegeben werden, etwa:

```
gcc teil1.c teil2.o teil3.o<cr>
```

wobei der Quelltext `teil1.c` erneut übersetzt und die hieraus entstehende (temporäre) Objektdatei mit den angegebenen anderen beiden Objektdateien zum fertigen Programm zusammengebunden wird. Hierbei wird die Objektdatei zu `teil1.c` wie gesagt nur temporär erzeugt, so dass anschließend keine neue Version von `teil1.o` existiert! (Diese erhält man nur durch alleiniges Übersetzen mit der Compileroption `-c` — wobei dann allerdings keine Objektdateien als Argumente zum Compiler angegeben werden können!)

Durch die Technik, mehrere Quelltexte zur Erstellung eines Programms zu verwenden, können neuartige Fehler auftreten, welche beim Compilieren der einzelnen Quelltexte noch nicht festgestellt werden können (Compiler läuft ab, ohne eine Fehlermeldung auszugeben), sondern erst beim Zusammenbinden festgestellt werden:

- Hat man (aus Versehen) in verschiedenen Quelltexten gleichnamige Funktionen definiert, so ist das beim Übersetzen der einzelnen Quelltexte unerheblich, d.h. die Objektdateien werden fehlerfrei erstellt. Beim Zusammenbinden der Objektdateien merkt jedoch der Binder, dass die besagte Funktion mehrfach definiert ist, gibt eine entsprechende Fehlermeldung aus und es kann kein fertiges Programm erstellt werden!
- Hat man in einem Quelltext eine Funktion deklariert und verwendet, aber vergessen, diese in einem anderen Quelltext zu definieren, so läuft ebenfalls das Erstellen der einzelnen Objektdateien fehlerfrei ab. Beim Zusammenbinden der Objektdateien merkt jedoch der Binder, dass die besagte Funktion nicht definiert ist, gibt eine entsprechende Fehlermeldung aus und es kann kein fertiges Programm erstellt werden!

Derartige Fehler heißen *Linkerfehler*.

Darüberhinaus erschließt sich ein weiteres Feld von möglichen Fehlern, die weder Compiler noch Linker bemerken, sondern erst zur Laufzeit zu inkonsistenten Daten und Programmabstürzen führen können:

Deklariert und verwendet man eine Funktion in einem Quelltext und definiert man sie in einem anderen, wobei zwischen Deklaration und Definition eine Diskrepanz besteht (etwa unterschiedliche viele Argumente bzw. Parameter oder andere Typen), so ist für den Compiler alles in Ordnung, da er den korrekten Aufruf ja nur anhand der Deklaration überprüfen kann und zwischen Deklaration und Aufruf ja keine Diskrepanz besteht — und für den Linker ist auch alles in Ordnung, da er ja nur nach einer Funktion mit dem entsprechenden Namen sucht und diese ja auch findet (leider nur vom falschen Typ — der Linker erkennt den Typ einer Funktion nicht!).

6.8.2 Eigene Header-Dateien

Besteht ein Programm aus mehreren Quelltexten, ist eine beteiligte Funktion in einem Quelltext definiert und wird diese in allen anderen Quelltexten verwendet, so muss diese Funktion bekanntlich in allen anderen Quelltexten jeweils deklariert werden, d.h. die Deklarationszeile muss mehrfach (mindestens einmal pro Quelltext, in dem die Funktion verwendet wird) eingetippt werden.

Zur Einsparung von Tipparbeit (und auch zur Vermeidung von den im letzten Unterabschnitt beschriebenen Diskrepanzen zwischen Funktionsdeklaration und -definition sinnvoll) ist folgende Technik:

- Die Deklarationszeile der (aller) in mehreren Quelltexten verwendeten Funktion(en) wird in eine separate Datei, etwa mit dem Namen **program.h**, geschrieben.
- Diese eigene Header-Datei (**program.h**) wird am Anfang jedes Programm-Quelltextes "includet".

Hierzu kann folgende Form der `include`-Präprozessoranweisung

```
#include "program.h"
```

verwendet werden. Diese Form mit den doppelten Anführungszeichen unterscheidet sich von der bei Bibliotheks-Header-Dateien üblichen Form mit den spitzen Klammern dadurch, dass beim `#include <...>` in speziellen Systemverzeichnissen nach der entsprechenden Header-Datei gesucht wird, und beim `#include "..."` zunächst im aktuellen Verzeichnis — da, wo auch der Quelltext steht — und wenn sie dort nicht gefunden wird, dann auch in den üblichen Systemverzeichnissen!

In einfachen Beispielen reicht i. Allg. eine eigene Header-Datei aus. Es macht auch nichts, wenn in der Header-Datei eine Funktion deklariert wird, welche im aktuellen Quelltext gar nicht verwendet wird (aber in einem anderen).

Zu einem komplexeren Programm kann man mehrere eigene Header-Dateien definieren, pro Gruppe von sinnzusammenhängenden Funktionen jeweils eine (Modularisierung).

Aus dem gesagten sollte ersichtlich sein, dass in eine Header-Datei nur Deklarationen und keine Definitionen von Funktionen gehören.

Eine in einer Header-Datei stehende Funktionsdefinition bewirkt, dass, wenn man diese in unterschiedliche Quelltexte eines Programms includet, diese Funktion mehrfach definiert ist und der Linker beim Zusammenbinden des fertigen Programms Schwierigkeiten bekommt.

7 Verschiedenes

7.1 Argumente und Funktionsergebnis von main.

Wie bereits erwähnt, wird das Hauptprogramm `main` selbst auch als Funktion aufgefasst, die vom Betriebssystem aufgerufen wird.

Einige Betriebssysteme unterstützen die vom C-Standard vorgesehene Übergabe von Argumenten vom Betriebssystem an das C-Programm bei dessen Aufruf und bei dessen Ende die Rückgabe von Funktionsresultaten vom C-Programm ans Betriebssystem.

7.1.1 Funktionsergebnis von main

Die Rückgabe eines Funktionsergebnisses erfolgt wie bei Funktionen üblich mittels der `return`-Anweisung. In der Regel wird nur `int` als Rückgabetypp eines C-Programms unterstützt. D.h. die `main`-Funktion kann wie folgt definiert werden:

```
int main(void)
{
    .
    .
    return 7;
}
```

Der bei `return` angegebene (ganzzahlige) Wert wird an das Betriebssystem zurückgegeben.

Auf UNIX-Rechnern wird dieser Ergebniswert in einer entsprechenden Shell-Variablen abgespeichert (bei einer Korn-Shell hat diese Variable den Namen `$?`).

Auf MS-DOS-Rechnern gibt es hierzu die Systemvariable `errorlevel`!

7.1.2 Argumente der Kommandozeile

Zur Übergabe von Argumenten an das C-Programm sieht der Standard vor, dass das Hauptprogramm wie folgt definiert werden kann:

```
int main(int argc, char *argv[])
{ ... }
```

Der ganzzahlige Parameter `argc` (argument-count) erhält als Wert die Anzahl der Argumente beim Programmaufruf und `argv` (argument-vector) kann als Feld der Länge `argc+1` von `char`-Adress-Variablen (oder auch als ein Zeiger auf Zeiger auf `char`) aufgefasst werden.

Gewöhnlich wird ein ausführbares Programm durch die Angabe seines Namens gestartet. Schreibt man nun hinter diesen Programmnamen durch Leerzeichen getrennt weitere Zeichenketten (welche natürlich keine Sonderzeichen des Betriebssystems enthalten dürfen), so werden der Programmname und diese Zeichenketten mittels des Adress-Vektors `argv` dem C-Programm verfügbar gemacht.

Sei etwa `echo` ein ausführbares C-Programm, dessen Funktion `main` durch

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

definiert wurde, so ist nach dem Aufruf

```
echo hello, world <RETURN>
```

folgende Situation gegeben:

`argc` hat den Wert 3 (= Anzahl der „Wörter“ beim Programmaufruf)

`argv[0]` ist die Adresse des ersten Wortes des Programmaufrufs, also die des (irgendwo im Maschinenspeicher abgelegten) Wortes „`echo`“,

`argv[1]` die des zweiten Wort des Programmaufrufs, also die von „`hello`,“ (mit dem Komma) und

`argv[2]` die des dritten Wortes „`world`“.

`argv[3]` (= `argv[argc]`) hat den Wert `NULL`, der anzeigt, dass keine weiteren Argumente mehr vorliegen.

Ruft man das gleiche Programm mit der Zeile

```
echo hello , world <RETURN>
```

auf, so ist `argc` gleich 4, `argv[0]` zeigt auf den Namen „`echo`“ des Programms,

`argv[1]` zeigt auf das Wort „`hello`“, `argv[2]` zeigt auf das Wort „`,`“, `argv[3]` zeigt auf das Wort „`world`“ und `argv[4]` (= `argv[argc]`) hat wiederum den Wert `NULL`.

Die beim Programmaufruf angegebenen Zeichenketten sind somit im Programm verfügbar und man kann diese wie sonstige Zeichenketten auch verwenden, etwa ausgeben:

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

```
int main(int argc, char *argv[])
```

```
{
```

```
    int i,
```

```
    for ( i = 1; i <argc; ++i)
```

```
        printf("%s ", argv[i]);
```

```
    return 0;
```

```
}
```

Bei vielen UNIX-Kommandos ist es möglich, das Kommando auf unterschiedliche Art und Weise aufzurufen (ohne oder mit einem Argument — ohne oder mit Parameter). Dies kann für eigene C-Programme durch Argumente der Kommandozeile auch erreicht werden. Ein Beispiel, in dem u.a. Optionen und Argumente erkannt und bearbeitet werden, ist in `fortschr/argument/n_sort4.c` nachzulesen.

Einige Betriebssysteme unterstützen die Übergabe weiterer Argumente (etwa Umgebungsvariablen) an C-Programme.

7.2 Aufzählungstypen

Ein Aufzählungstyp ist eine Folge von ganzzahligen Werten, wobei jeder dieser Werte einen Namen erhält.

Die Definition eines Aufzählungstypen kann überall dort erfolgen, wo auch eine Funktion deklariert werden können, also i. Allg. zu Beginn des Anweisungsteils einer Funktion oder extern, außerhalb jeder Funktion. Ihre Definition geschieht wie folgt:

Schlüsselwort **enum**, optional gefolgt von einem neuen Namen (Bezeichner), anschließend öffnende geschweifte Klammer, durch Komma getrennte Liste von (neuen) Namen, schließende geschweifte Klammer und Strichpunkt. Die in der Liste stehenden Namen haben von 0 beginnend aufsteigend durchnummerierte ganzzahlige Werte.

Beispiel:

```
enum Farbe { rot, gruen, blau};
```

Hierdurch ist ein neuer Typ definiert, dieser Typ hat den Namen **enum Farbe** (Typname besteht aus zwei Worten). Die beteiligten Konstanten (Aufzählungskonstanten) haben folgende Werte: **rot** hat den Wert 0, **gruen** hat den Wert 1 und **blau** hat den Wert 2.

Im folgenden Programmtext können die drei Konstanten **rot**, **gruen** und **blau** wie ganzzahlige Konstanten (mit den entsprechenden Werten) verwendet werden — können einer ganzzahligen Variablen zugewiesen werden, mit ihnen kann gerechnet werden — aber es ist auch folgende Anwendung möglich:

```
...
enum Farbe { rot, gruen, blau};
...
int farbe;
...
if ( farbe == blau )
    {...}
...
```

d.h. es kann mit den Namen im Aufzählungstyp gearbeitet werden, ohne den dahinterstehenden ganzzahligen Wert zu kennen.

Darüberhinaus können auch Variablen vom Aufzählungstyp vereinbart werden:

```
...
enum Farbe { rot, gruen, blau};
...
enum Farbe farbe;
...
if ( farbe == blau )
    {...}
...
```

Doch auf den heutigen Compilern ist dies gleichwertig zur Definition einer gewöhnlichen ganzzahligen Variablen, d.h. dieser Variablen **farbe** könnte man den Wert -25

zuweisen, ohne eine Fehlermeldung zu erhalten.

Gibt man dem Aufzählungstypen bei seiner Definition keinen Namen, so sind nur die Konstanten in der Liste bekanntgemacht und man kann evtl. nur bei der Definition des Aufzählungstypen Variablen von diesem Typ definieren, etwa :

```
enum { rot, gruen, blau} farbe;
```

Will man später auf den Typen (nicht nur auf die beteiligten Konstanten) zugreifen, muss man hinter **enum** einen Namen angeben.

Wird ein Aufzählungstyp innerhalb einer Funktion (also zu Beginn deren Anweisungsteils) definiert, so ist der Aufzählungstyp und jede mit diesem verbundene Konstante nur innerhalb der Funktion bekannt.

Wird der Aufzählungstyp extern, also außerhalb jeder Funktion definiert, so ist der Aufzählungstyp und jede mit diesem verbundene Konstante von seiner Definition an im restlichen Quelltext bekannt.

Will man einen Aufzählungstyp als extern deklarieren, so muss man dies am Anfang des Programms (vor der ersten Funktion) tun.

Weiterhin kann man `typedef` dazu verwenden, Programme so zu schreiben, dass man sie leichter von einer Maschine auf eine andere portieren kann.

Will man etwa einen ganzzahligen Datentypen haben, der auf allen Rechnern einen Speicherbereich von 4 Byte haben soll, so kann man in seinem Programm etwa einen (an sich nicht existierenden) Datentypen `Int` verwenden und auf Rechnern, auf denen ein gewöhnliches `int` 4 Byte Speicher belegt, vor der Verwendung von `Int` diesen Typen mittels

```
typedef int Int;
```

auf das gewöhnliche `int` abbilden, und auf anderen Rechnern, auf denen ein gewöhnliches `int` nur 2 Byte Speicher belegt, ein `long` jedoch 4 Byte belegt, den Typen `Int` mittels

```
typedef long Int;
```

auf diesen abbilden. Bei einer Portierung auf einen anderen Rechner ist somit nur diese eine Zeile mit dem `typedef` zu ändern!

Nochmals zur Verdeutlichung: mittels `typedef` kann man keine wirklich neuen Typen definieren, sondern nur einem vorhandenen Typen einen neuen Namen geben.

Wird ein `typedef` innerhalb einer Funktion (dann zu Beginn deren Anweisungsteils) erwendet, so ist der neu definierte Typ nur innerhalb der Funktion bekannt. Wird `typedef` extern, also außerhalb jeder Funktion verwendet, so ist der definierte Typ von seiner Definition an bis ans Ende des Quelltextes bekannt.

Setzt sich ein Programm aus mehreren Quelltexten zusammen und soll ein selbstdefinierter Typ in jedem Quelltext verwendet werden, so muss dieser Typ in allen Quelltexten mittels `typedef` definiert werden. (Bzw. die `typedef`-Definition ist in eine Header-Datei zu schreiben und diese überall wo nötig einbinden!)

7.4 Das Schlüsselwort `const`

Variablen, Objekte oder Werte eines gewissen Types können die zusätzliche Qualifikation `const` haben.

Hierzu muss i. Allg. bei der Erzeugung der Variablen direkt vor oder direkt hinter dem Typnamen das Schlüsselwort `const` aufgeführt werden!

Ein `const`-Wert bzw. eine `const`-Variable behält während seiner bzw. ihrer Lebenszeit den bei der Erzeugung gegebenen Wert (bzw. soll diesen Wert behalten und ein ordentlicher Compiler muss hierfür sorgen!).

Bei der Definition einer gewöhnlichen `const`-Variablen muss man dafür Sorge tragen, dass sie bei ihrer Erzeugung den ihr zugedachten Wert bekommt. D.h. man muss sie explizit initialisieren, etwa:

```
const double pi = 3.1415926;    oder  
double const pi = 3.1415926
```

Diese "Variable" `pi` wird im Folgenden wie eine "Konstante" behandelt, darf also nicht mehr verändert werden — kann etwa auch nicht mehr auf der linken Seite einer Zuweisung stehen oder durch Einlesen einen neuen Wert bekommen:

```
pi = x + y;          /* Fehler: pi ist const */
scanf("%lg", &pi);  /* Fehler: pi ist const */
```

Auch Funktionsparameter können als **const** vereinbart werden (bei Funktionsdefinition und Funktionsdeklaration erforderlich!):

```
void f ( const int );    /* Funktionsdeklaration */
...
void f ( const int i)    /* Funktionsdefinition */
{ ... }
```

Beim Funktionsaufruf wird der Parameter als lokale Variable erzeugt und erhält als Wert wie üblich den Wert des entsprechenden Funktionsargumentes des Aufrufes (Funktionsparameter können nicht explizit initialisiert werden, die Initialisierung wird anhand des Funktionsargumentes vorgenommen!) und diese Variable behält während ihrer Lebenszeit (Dauer des Funktionsaufrufes) diesen Wert. Diese Variable ist also während des Funktionsaufrufes konstant.

Dies hat i. Allg. für den Aufrufer der Funktion keine Bewandtnis, er kann die Funktion mit einer Konstanten oder Variablen als Argument aufrufen. Der Wert des Funktionsargumentes — ob dieses nun konstant ist oder nicht — wird (als Wert) in die lokale Adress-Variable der Funktion kopiert. Dass diese lokale Variable konstant ist und im Lauf des Funktionsaufrufs nicht geändert wird, merkt der Aufrufer gar nicht, da diese Variable nach dem Funktionsaufruf gar nicht mehr vorhanden ist!

7.4.1 Adress-Variablen und const

Bei Adress-Variablen muss unterschieden werden:

- ist die Adress-Variable selbst **const** oder
- ist das **const**, worauf die Adress-Variable zeigt?

Im ersten Fall sieht die Definition der Adress-Variable wie folgt aus:

```
char wort[64];
char * const p = wort;
```

Diese Definition ist wie folgt (von rechts nach links) zu lesen: **p** ist ein konstanter (Schlüsselwort **const**) Zeiger (*) auf ein **char**.

Als konstantes Objekt muss bei der Erzeugung ein initialisierender Wert (hier die Adresse des Feldanfangs des Feldes **wort**) angegeben werden! Das, worauf die konstante Adress-Variable zeigt, kann durchaus verändert werden — ist also nicht konstant:

```
*p = 'H';
```

Im zweiten Fall sieht die Definition der Adress-Variablen so aus:

```
char const *p;
```

(Die Reihenfolge von **char** und **const** spielt keine Rolle!) Diese Definition ist wie folgt (wieder von rechts nach links) zu lesen: **p** ist ein Zeiger (*) auf ein konstantes (Schlüsselwort **const**) **char**, also etwa ein Zeiger auf eine konstante Zeichenkette.

Das, worauf gezeigt wird, soll konstant sein, nicht jedoch der Zeiger selbst. Entsprechend braucht `p` nicht explizit initialisiert zu werden, es können im Laufe des Programms unterschiedliche Adressen — aber jeweils von konstanten Zeichenketten — dem `p` zugewiesen werden:

```
char wort[64];
char const name[] = "Hanrath"; /* name ist ein Feld von konstanten Zeichen */
char const *p;
...
p = "hallo"; /* ok, "hallo" ist konstante Zeichenkette */
p = "Welt"; /* ok, "Welt" ist konstante Zeichenkette */
p = name; /* ok, name ist konstantes Zeichenfeld */
p = wort; /* Fehler: wort ist keine konstante Zeichenkette */
...
```

Natürlich gibt es auch konstante Zeiger auf konstante Objekte. Die Definition sieht dann so aus:

```
char const * p const = "hallo";
```

Hier kann weder `p` noch das, worauf `p` zeigt, verändert werden!

7.4.2 Zeiger auf `const` als Funktionsparameter

Zeiger auf konstante Objekte spielen bei Funktionsparametern eine bedeutende Rolle: Wird eine Adresse eines Objektes etwa vom Hauptprogramm an eine Funktion übergeben, so muss (sollte) das Hauptprogramm wissen, ob das Objekt, dessen Adresse übergeben wird, von der Funktion abgeändert wird oder nicht!

Wird das Objekt nicht abgeändert, so kann die Funktion gefahrlos auch mit der Adresse eines konstanten Objektes aufgerufen werden.

Wird das Objekt jedoch durch die Funktion abgeändert, so darf die Funktion nicht mit der Adresse eines konstanten Objektes aufgerufen werden!

Dies sollte man bei Funktionsdeklaration und Definition deutlich machen, etwa:

- Funktion ändert eine übergebene Zeichenkette nicht:

```
void g ( const char *); /* Deklaration */
...
void g ( const char *s) /* Definition */
{ ... /* *s wird nicht geaendert */ }
...
int main(void)
{ char wort[64];
  ...
  g(wort); /* ok, wort zwar variabel, wird aber nicht geaendert */
  g("Hallo"); /* ok, "hallo" ist konstant */
  ...
}
```

- Funktion ändert eine übergebene Zeichenkette:

```
void f ( char *);    /* Deklaration */

...
void f ( char *s)    /* Definition */
{ ... /* *s wird geaendert */ }
...
int main(void)
{ char wort[64];
  ...
  f(wort);           /* ok, wort zwar variabel, wird durch f geaendert */
  f("Hallo");        /* Fehler, "hallo" ist konstant */
  ...
}
```

Dieser Fehler wird leider jedoch von einigen Compilern nicht erkannt, was ggf. zu Laufzeitfehlern führt!

7.5 Das Schlüsselwort `volatile`

Variablen, Objekte oder Werte eines gewissen Types können die zusätzliche Qualifikation `volatile` haben.

Auch hier muss das Schlüsselwort `volatile` i. Allg. bei der Erzeugung der Variablen direkt vor oder direkt hinter dem Typnamen aufgeführt werden!

Die Bedeutung von `volatile` ist die, dass das entsprechende Objekt spezielle Eigenschaften hat, die vom Compiler bei einer eventuellen Optimierung berücksichtigt werden müssen.

Der ANSI-C-Standard enthält keine genaueren Vorgaben bezüglich Optimierung und Behandlung von `volatile`-Variablen.

8 Operatoren, Ausdrücke und Anweisungen

8.1 Operatoren

C besitzt eine relativ reichhaltige Menge von Operatoren. Mitunter steht ein Zeichen für mehrere Operatoren, meist einmal für einen unären Operator und einmal für einen binären Operator. Die jeweilige Bedeutung ist dann aus dem Kontext zu entnehmen.

Folgende Tabelle gibt eine Übersicht über alle in C verfügbaren Operatoren.

Operatoren auf gleicher Höhe haben die gleiche Vorrangstufe (Priorität), weiter oben stehende Operatoren haben höhere Priorität als weiter unten stehende.

Zu Operatoren mit gleicher Vorrangstufe ist die sogenannte „Assoziativität“ angegeben, welche festlegt, in welcher Reihenfolge die Auswertung eines Ausdrucks mit gleichrangigen Operatoren erfolgt. (Etwa: beim binären `-` (Subtraktion) wird der Ausdruck `a-b-c` in der Reihenfolge `(a-b)-c` (von links nach rechts) ausgewertet.)

Der Programmierer kann natürlich durch Klammerung (runde Klammern) die Reihenfolge der Auswertung beeinflussen.

Operator	Assoziativität
<code>() [] -> .</code>	von links nach rechts
<code>! ~ ++ -- + - * & (type) sizeof</code>	von rechts nach links
<code>* / %</code>	von links nach rechts
<code>+ -</code>	von links nach rechts
<code><< >></code>	von links nach rechts
<code>< <= > >=</code>	von links nach rechts
<code>== !=</code>	von links nach rechts
<code>&</code>	von links nach rechts
<code>^</code>	von links nach rechts
<code> </code>	von links nach rechts
<code>&&</code>	von links nach rechts
<code> </code>	von links nach rechts
<code>?:</code>	von rechts nach links
<code>= += -= *= /= %= &= ^= = <<= >>=</code>	von rechts nach links
<code>,</code>	von links nach rechts

Tabelle 6: C-Operatoren

In sinngemäß zusammenhängenden Gruppen folgt die Bedeutung der einzelnen Operatoren. Die Zeilenangaben beziehen sich auf die Zeilen der obigen Tabelle.

Arithmetische Operatoren

+	-	(zweite Zeile) unäre Vorzeichenoperatoren.
*	/	(dritte Zeile) binär, Multiplikation und Division.
%		binär, Modulus-Operator, definiert nur für ganzzahlige Operanden, $a\%b$ liefert den Rest der Division von a durch b .
+	-	(vierte Zeile) binär, Addition und Subtraktion.

Diese Operatoren sind für arithmetische Operanden definiert. Zu beachten ist, dass bei ganzzahligen Operanden das Ergebnis ebenfalls ganzzahlig ist, z.B. liefert die Division $1/2$ das ganzzahlige Ergebnis 0!

Darüberhinaus ist das Ergebnis der Verknüpfung von zwei **unsigned**-Operanden wieder **unsigned**, so liefert die Subtraktion $3u - 5u$ ein ziemlich großes, positives Ergebnis, und nicht wie erwartet -2 .

Bei den Operatoren $/$ und $\%$ ist das Resultat für negative ganzzahlige Operanden maschinenabhängig.

Relationale und logische Operatoren

In C gibt es keinen eigenen Datentyp zur Speicherung der logischen Werte *wahr* und *falsch*. Jeder arithmetische Wert (und auch ein Adress-Wert) kann als Wahrheitswert interpretiert werden:

Ist der Wert ungleich 0, so wird er als *wahr* interpretiert, ist er gleich 0, so wird er als *falsch* interpretiert.

Das Ergebnis eines relationalen oder logischen Ausdrucks ist 1, falls die Relation zutrifft bzw. der logische Ausdruck *wahr* ist, und 0 sonst.

relationale Operatoren:

<, >	binär, Test auf kleiner bzw. größer. Z.B. ist $a < b$ genau dann <i>wahr</i> (=1), wenn der arithmetische Operand a kleiner als der Operand b ist.
<=, >=	binär, Test auf kleiner gleich bzw. größer gleich.
==, !=	binär, Test auf gleich bzw. ungleich.

Man beachte, dass die Priorität von $==$ und $!=$ geringer als die der übrigen relationalen Operatoren ist!

logische Operatoren:

!	unär, logische Negation; aus einem arithmetischen Wert ungleich 0 wird der Wert 0, aus dem Wert 0 wird der Wert 1.
&&	binär, logische <i>UND</i> -Verknüpfung.
	binär, logische <i>ODER</i> -Verknüpfung.

Der Vorrang von $\&\&$ ist höher als der von $\|\|$. Die Bewertung eines logischen Ausdrucks (von links nach rechts) hört auf, sobald der Wahrheitswert des ganzen Ausdrucks feststeht.

Etwa: $(a < b \ \&\& \ a < f(b))$

Es wird zunächst überprüft, ob a kleiner als b ist. Ist dies nicht der Fall, so ist der ganze logische Ausdruck unabhängig vom Wahrheitswert der rechten Seite *falsch*. Die

rechte Seite braucht nicht (und wird nicht) ausgewertet zu werden. Die Funktion `f` mit ihren möglichen Nebeneffekten wird also gar nicht erst aufgerufen!

Bitweise Operatoren:

Die Operatoren zur Manipulierung einzelner Bits dürfen nur auf ganzzahlige Operanden (`char`, `short`, `int`, `long` jeweils `signed` oder `unsigned`) angewendet werden.

<code>&</code>	(achte Zeile) binär, bitweise <i>UND</i> -Verknüpfung.
<code> </code>	binär, bitweise inklusive <i>ODER</i> -Verknüpfung.
<code>^</code>	binär, bitweise exklusive <i>ODER</i> -Verknüpfung.
<code><< >></code>	binär, Shift nach links bzw. Shift nach rechts. Verschiebt das Bitmuster des linken Operanden um soviel Stellen nach links bzw. rechts, wie der rechte Operand angibt. Es wird gewöhnlich mit Nullen aufgefüllt. Ausnahme: bei einigen Implementierungen wird beim Shift nach rechts von vorzeichenbehafteten Typen das Vorzeichen nachgeschoben. Das Ergebnis ist nicht definiert, falls der rechte Operand negativ oder größer als die Länge des linken Operanden ist.
<code>~</code>	unär, (Einer-)Komplement-Operator.

(Man beachte, dass bei den Operatoren `<<`, `>>` und `~` die Operanden nicht verändert werden, sondern die Änderung nur als Ergebniswert des entsprechenden Ausdrucks vorliegt.)

Inkrement- und Dekrement-Operatoren

Die unären Operatoren `++` und `--` können nur auf Variablen (genauer: l-values: Ausdrücke, welche auf der linken Seite einer Zuweisung stehen können) von arithmetischem Typ angewendet werden.

Der Inkrement-Operator `++` erhöht den Wert der entsprechenden Variablen um 1, der Dekrement-Operator `--` subtrahiert von der Variablen den Wert 1.

Beide Operatoren kommen in Postfix- und Präfix-Form vor.

Steht ein Inkrement- oder Dekrement-Operator in einem Ausdruck hinter seinem Operanden (Postfix), so wird der alte Wert des Operanden im Ausdruck verwendet und anschließend der Wert des Operanden um 1 abgeändert (addiert bei `++` bzw. subtrahiert bei `--`).

Steht der Operator vor seinem Operanden in einem Ausdruck (Präfix), so wird erst der Wert des Operanden verändert und dann der neue Wert im Ausdruck verwendet.

Beispiele:

```
int s[100], i;
i = 7;
i++;           /* i ist jetzt 8 */
s[i++] = 27;   /* s[8] wird 27 und i ist danach 9 */
s[++i] = 5;    /* i wird 10 und s[10] wird 5 */
```

Präfix- und Postfixform dieser Operatoren haben also unterschiedliche Seiteneffekte. Inkrement- und Dekrement-Operatoren sind insbesondere dann mit Vorsicht zu genießen, wenn die Variable, auf die sie angewendet werden, mehrfach im Ausdruck

vorkommt, etwa:

```
x = x++ + ++x ;           /* ????   schlechter Stil! */
```

Zuweisungsoperatoren

Die einfachste Form der Zuweisung geschieht mit dem Gleichheitszeichen, etwa

```
x = y + 1;    /* x erhaelt den Wert der Summe aus y und 1 */
```

(Das Semikolon gehört nicht zum Zuweisungsausdruck, sondern macht aus diesem eine Anweisung!)

Jeder Zuweisungsausdruck besitzt einen Typ und einen Wert, nämlich Typ und Wert der linken Seite nach der Zuweisung.

Beispiel: `x = f( y = z );`

Die Variable `y` erhält den Wert der Variablen `z`, der Ausdruck `y = z` hat den Typ und Wert der Variablen `y` nach der Zuweisung und die Funktion `f` wird mit dem Wert dieses Zuweisungsausdrucks als Argument aufgerufen. Die Variable `x` erhält als Wert das Funktionsergebnis.

Da Zuweisungen von rechts nach links ausgewertet werden (vgl. obige Tabelle), sind z.B. Ausdrücke folgender Form möglich:

```
double y, z;
y = z = 2 + 3;
```

`z` erhält den Wert 5. Der Wert des Ausdrucks `z = 2 + 3` ist gleich dem `double`-Wert 5, und dieser Wert wird der Variablen `y` zugewiesen. Somit erhalten `y` und `z` den Wert 5.

Ist op einer der binären Operatoren `+`, `-`, `/`, `*`, `%`, `<<`, `>>`, `&`, `^` oder `|` und sind `x` und `y` zulässige Operanden für den Operator, so ist

`x op=y`
eine abkürzende Schreibweise für

```
(x) = (x) op (y)
```

mit der Einschränkung, dass der Operand `x` nur einmal ausgewertet wird (Seitenefekte).

Etwa

```
int i[5], j = 0;

i[0] = 5;
i[j++] += 2;           /* i[0] erhaelt den Wert 7
                        und j wird 1 */

j = 0;
i[j++] = i[j++] + 3;    /* i[0] wird um drei erhoeht (???)
                        und j wird zweimal inkrementiert */
```

Eine derartige Zuweisung hat wiederum den Typ und Wert der linken Seite nach der Zuweisung.

Zuweisungen ganzer Felder (z.B. Zeichenketten) ist nicht vorgesehen. (Hier muss man die Zuweisung elementweise durchführen oder als Funktionen realisieren.)

Die restlichen Operatoren

Funktionsaufruf: ()

etwa: `printf(" hello, world \n");`

Näheres zu Funktionen ist bereits im Kapitel 6 über Funktionen erläutert worden.

Feldindizierung: []

etwa: `s[10] = 0;`

Zugriff auf Komponenten einer Struktur: . oder ->

Strukturen werden später behandelt (siehe Kapitel 12).

Adress-Operator: &

Ist im Abschnitt 6.6.1 bereits behandelt worden.

Verweis-Operator: *

Ist im Abschnitt 6.6.3 bereits behandelt worden.

Cast-Operator:

Bei der Auswertung von Ausdrücken werden nach gewissen Regeln implizit die Typen der Operanden angeglichen. Der Cast-Operator bietet die Möglichkeit, den Typ eines Ausdrucks explizit in einen anderen Typ umzuwandeln (erzwungene Typumwandlung!). Form eines Cast-Ausdrucks:

`( Typ ) Ausdruck`

Der *Ausdruck* wird ausgewertet und das Ergebnis (falls möglich) in den in Klammern angegebenen *Typ* umgewandelt.

Der Cast-Operator findet hauptsächlich bei der Umwandlung von Adress-Werten Verwendung, etwa:

```
double *f,g;
char *c;
f = &g;
c = (char *) f;
```

Zunächst erhält die `double`-Adress-Variable `f` die Adresse der `double`-Variablen `g` (der Typ dieser Adresse ist: Adresse eines `double`-Objektes!). Anschließend erhält die `char`-Adress-Variable als Wert ebenfalls diese Adresse, jedoch mit dem Typ: Adresse eines `char`-Objektes.

Auf diese Weise kann man z.B. den Inhalt einer `double`-Variablen als Zeichenkette ausgeben (um etwa die interne Abspeicherung von Gleitkommazahlen herauszubekommen).

sizeof-Operator

Der `sizeof`-Operator wird in zwei Formen angewendet, auf Objekte (keine Funktionen!): `sizeof Objekt`

oder auf Typen: `sizeof (Typ)`

Er liefert als Ergebnis die Größe vom angegebenen *Objekt* bzw. eines Objektes vom angegebenen *Typ* in Bytes. Das ganzzahlige Ergebnis hat den Typ `size_t`, der je nach Implementation `unsigned int` oder `unsigned long` ist. `size_t` wird mittels `typedef` in der Header-Datei `<stddef.h>` definiert.

Der `sizeof`-Operator liefert die tatsächlich implementierten Größen der Typen und Objekte:

Auf Maschinen, deren `int`-Werte in 2 Byte abgespeichert werden, liefert der Ausdruck `sizeof (int)` den Wert 2.

Auf Maschinen, deren `int`-Werte in 4 Byte abgespeichert werden, liefert derselbe Ausdruck `sizeof (int)` den Wert 4.

Der `? :-`-Operator

Der `? :-`-Operator dient zur Konstruktion bedingter Ausdrücke.

Bedingte Ausdrücke haben die Form

Ausdruck1 `? Ausdruck2 : Ausdruck3`

Es wird zunächst der *Ausdruck1* ausgewertet. Ist das Ergebnis dieses Ausdrucks ungleich 0 (d.h. *Ausdruck1* wird als *wahr* interpretiert!), so wird *Ausdruck2* ausgewertet und das Ergebnis des gesamten bedingten Ausdrucks ist das Ergebnis von *Ausdruck2* nach seiner Auswertung.

Ist das Ergebnis von *Ausdruck1* gleich 0 (d.h. *Ausdruck1* wird als *falsch* interpretiert), so wird *Ausdruck3* ausgewertet und das Ergebnis des ganzen bedingten Ausdrucks ist der Wert von *Ausdruck3*.

Beispiel:

```
z = ( a > b ) ? a : b ;
```

Effekt: *z* wird als Wert das Maximum von *a* und *b* zugewiesen.

Sollten *Ausdruck2* und *Ausdruck3* verschiedene Typen besitzen, so wird, damit der Typ des bedingten Ausdruck eindeutig festliegt, unabhängig vom Ergebniswert von *Ausdruck1* der Ergebniswert der Auswertung von *Ausdruck2* bzw. *Ausdruck3* entsprechend der Umwandlungsregeln in den größeren der beiden Typen (von *Ausdruck2* und *Ausdruck3*) oder einen gemeinsamen größeren umgewandelt.

Der Komma-Operator

Der Komma-Operator erlaubt es, zwei Ausdrücke (im geschachtelten Fall mehrere) zu einem Ausdruck zusammenzufassen:

Ausdruck1 , *Ausdruck2*

Ausdruck1 wird ausgewertet, das Ergebnis des Ausdrucks vergessen, anschließend wird *Ausdruck2* ausgewertet und das Ergebnis des gesamten Komma-Ausdrucks hat Typ und Wert von *Ausdruck2* nach seiner Auswertung.

Es werden somit nur die Effekte (etwa Zuweisungen oder Inkrementierungen von Variablen) und nicht das Ergebnis von *Ausdruck1* benötigt.

Durch Komma-Ausdrücke hat man die Möglichkeit, mehrere Ausdrücke an eine Stelle zu plazieren, an der C genau einen Ausdruck verlangt (etwa zur Initialisierung und

Inkrementierung von `for`-Schleifen).

Ein Wort zu den Operanden.

Die Operanden der meisten Operatoren sind arithmetische Operanden. Hiermit sind zunächst Operanden der üblichen arithmetischen Typen `char`, `int`, `float` und `double` (in ihren sämtlichen Formen inklusive der `enum`-Form von `int`) gemeint.

In Ergänzung zu diesen arithmetischen Operanden lassen einige arithmetische Operatoren eingeschränkt auch Adressen bzw. Adress-Variablen als Operanden zu. Die hierdurch mögliche Zeigerarithmetik wird später behandelt (Kapitel 13).

8.2 Ausdrücke

Ausdrücke sind wie üblich und bereits erwähnt Verknüpfungen von Operatoren mit Operanden.

Ausdrücke besitzen (in der Regel!) einen Typ und einen Wert. Ausnahmen sind Ausdrücke vom Typ `void`, welche keinen Wert besitzen (etwa: Aufruf einer Funktion mit Ergebnistyp `void`).

Konstante Ausdrücke sind Ausdrücke, deren Operanden Konstanten sind und welche keine Inkrement/Dekrement-Operatoren, keine Zuweisungsoperatoren, keine Funktionsaufrufe und keinen Kommaoperator enthalten.

Die Werte konstanter Ausdrücke stehen somit zur Compilierzeit bereits fest. Derartige Ausdrücke werden in C an verschiedenen Stellen benötigt, etwa: Dimensionierung von Vektoren, Werte in Aufzählungstypen, Initialisierung externer und statischer Variablen (vgl. Kapitel 9) etc..

Bei der Auswertung der Ausdrücke werden die in der Tabelle der Operatoren stehenden Prioritäts- und Assoziativitätsregeln befolgt.

Bei binären Operatoren legt C i. Allg. nicht fest, in welcher Reihenfolge die Operanden ausgewertet werden (Ausnahme: die logischen Operatoren `&&` und `||`; hier wird zunächst der linke und dann — nur falls erforderlich, vgl. Seite 139 — der rechte Operand ausgewertet!).

Sind etwa `f` und `g` Funktionen mit einem arithmetischen Rückgabetyt und ist `x` eine entsprechende Variable, so ist in

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

nicht festgelegt, welche der beiden Funktionen zuerst aufgerufen wird. (Durch die möglichen Seiteneffekte dieser Funktionen kann u.U. das Ergebnis der Addition von der Reihenfolge der Auswertung abhängen!)

Bei der Auswertung arithmetischer Ausdrücke, in die Operanden von unterschiedlichem Datentyp involviert sind, werden die Typen nach den im nächsten Abschnitt zusammengefassten Regeln implizit angeglichen.

8.3 Typumwandlungen

Da die darstellbaren Zahlenbereiche der arithmetischen Datentypen nicht vollständig vom Standard festgelegt sind, berücksichtigen die Umwandlungsregeln die möglichen Implementierungen dieser Typen.

Die Darstellung der Umwandlungsregeln erfolgt in drei Abschnitten. Der erste Abschnitt beschreibt die sogenannte „ganzzzahlige Aufwertung“, im zweiten Abschnitt wird behandelt, wie die Umwandlungen durchgeführt werden und der dritte Abschnitt beschreibt, wann derartige Umwandlungen implizit durchgeführt werden.

Neben diesen impliziten Typumwandlungen existieren natürlich die erzwungenen Typumwandlungen (mittels des Cast-Operators, siehe Seite 142).

8.3.1 Ganzzzahlige Aufwertung

„Kurze“ ganzzzahlige Werte (d.h. Werte vom Typ `char`, `short`, jeweils `signed` oder `unsigned`, sowie Werte vom Aufzählungstyp) dürfen überall in einem Ausdruck stehen, wo auch `int`-Objekte stehen dürfen.

Kommt in einem Ausdruck ein derartiger kurzer ganzzzahliger Wert vor und kann `int`-Wert alle Werte des „kurzen“ Types darstellen, so wird dieser „kurze“ Wert vor der Auswertung des Ausdrucks in `int` umgewandelt. Ansonsten wird dieser Wert in `unsigned int` umgewandelt.

Diese Umwandlung kurzer ganzzzahliger Werte vor der Auswertung eines Ausdrucks heißt auch „integer-Erweiterung“ oder auch „integral promotion“.

8.3.2 Wie wird umgewandelt?

- (a) Umwandlung ganzzzahliger Werte (`int`, `long`, jeweils `signed` oder `unsigned`) in einen vorzeichenlosen (`unsigned`) ganzzzahligen Typ:
Ist N die größte im vorzeichenlosen Typ, in den umgewandelt werden soll, darstellbare Zahl und i der Wert, der umgewandelt werden soll, so ist der umgewandelte Wert j die kleinste nichtnegative Zahl, die kongruent zu i modulo $(N + 1)$ ist.
- (b) Umwandlung ganzzzahliger Werte in vorzeichenbehaftete ganzzzahlige Typen:
Wenn der umzuwandelnde Wert im vorzeichenbehafteten Typ ohne Veränderung dargestellt werden kann, so bleibt der Wert unverändert. Ansonsten ist das Ergebnis implementationsabhängig.
- (c) Umwandlung ganzzzahliger Werte in Gleitkommatypen:
Liegt der ganzzzahlige Wert im darstellbaren Bereich des Gleitkommatypes (d.h. zwischen der kleinsten und der größten in diesem Typ darstellbaren Zahl), so ist das Resultat der nächst größere oder kleinere darstellbare Gleitkommawert (vom Standard nicht näher spezifiziert!).
Liegt der Wert nicht im darstellbaren Zahlenbereich, so ist das Ergebnis nicht definiert.

- (d) Umwandlung von Gleitkommawerten in ganzzahlige Typen:
Es wird zunächst der gebrochene Teil (Nachkommateil) des Gleitkommawertes (in exponentenfreier Darstellung) abgeschnitten. Ist das Resultat im ganzzahligen Typ darstellbar, so ist dieses Resultat der umgewandelte Wert.
Ansonsten ist die Umwandlung nicht definiert.
- (e) Umwandlung von Gleitkommawerten in (andere) Gleitkommatypen:
Wird ein weniger genauer Gleitkommawert in einen gleich oder höher genauen Typ umgewandelt, so wird der Wert nicht geändert.
Wird ein Gleitkommawert in einen weniger genauen Gleitkommatyp umgewandelt und liegt der umzuwandelnde Wert im darstellbaren Zahlenbereich des weniger genauen Types, so ist das Resultat wiederum die nächst größere oder nächst kleinere darstellbare Zahl. Ansonsten ist das Ergebnis nicht definiert.

8.3.3 Wann wird umgewandelt?

Die Regeln für die impliziten Typumwandlungen bei binären arithmetischen Operatoren zitiere ich aus der ANSI-Sprachbeschreibung:

Zuerst wird, wenn einer der beiden Operanden `long double` ist, der andere in `long double` umgewandelt.

Andernfalls wird, wenn einer der beiden Operanden `double` ist, der andere in `double` umgewandelt.

Andernfalls wird, wenn einer der Operanden `float` ist, der andere in `float` umgewandelt.

Andernfalls werden beide Operanden der Integer-Erweiterung unterworfen; wenn dann einer der beiden Operanden `unsigned long int` ist, wird der andere ebenfalls

in `unsigned long int` umgewandelt.

Andernfalls, wenn ein Operand `long int` und der andere `unsigned int` ist, hängt der Effekt davon ab, ob `long int` alle Werte von `unsigned int` darstellen kann. Falls ja, wird der eine Operand von `unsigned int` in `long int` umgewandelt; falls nein, werden beide in `unsigned long int` umgewandelt.

Andernfalls, wenn ein Operand `long int` ist, wird der andere in `long int` umgewandelt.

Andernfalls, wenn einer der Operanden `unsigned int` ist, wird der andere in `unsigned int` umgewandelt.

Andernfalls haben beide Operanden den Typ `int`.

8.4 Anweisungen

Die grundlegenden Anweisungen sind die sogenannten „expression-statements“. Ein expression-statement ist ein Ausdruck, hinter dem ein Semikolon steht (Ausdrucksanweisung). (Hier ist auch die leere Anweisung zugelassen, d.h. kein Ausdruck mit einem Semikolon dahinter!)

Durch Verbundanweisungen (*Compound-Statements*) werden mehrere Anweisungen zu einem Block zusammengefasst. Hierdurch hat man die Möglichkeit, mehrere Anweisungen an eine Stelle zu plazieren, in der C genau eine Anweisung verlangt.

Verbundanweisungen bestehen aus der öffnenden geschweiften Klammer, einer optionalen Liste von Vereinbarungen, einer optionalen Liste von Anweisungen sowie der schließenden geschweiften Klammer.

Der Anweisungsteil einer Funktion ist beispielsweise eine Verbundanweisung!

Wie bei der zu einer Funktion gehörenden Verbundanweisung können am Anfang einer jeden Verbundanweisung, etwa der zu einer **for**-Schleife, neue Variablen definiert oder auch Funktionen deklariert werden. Derartige Variablen und Funktionen sind nur innerhalb der Verbundanweisung bekannt. (Der Speicherbereich für solche Variablen wird am Ende der Verbundanweisung freigegeben, die Variable existiert danach nicht mehr!)

Anweisungen können markiert, d.h. mit einer Marke (Label) unmittelbar vor der Anweisung steht eine sogenannte Marke (Identifer) mit einem Doppelpunkt dahinter. Eine derartige Marke kann nun als Ziel einer Sprung-Anweisung (**goto** Marke;) verwendet werden.

Der Gültigkeitsbereich dieser Marke ist die aktuelle Funktion.

9 Speicherklassen

9.1 Speicherklassen von Variablen

Variablen können sich nicht nur vom Typ her unterscheiden, sondern auch in der sog. *Speicherklasse*.

Die Speicherklasse einer Variablen hängt davon ab, wo und wie die Variable definiert wird.

Variablen unterschiedlicher Speicherklassen unterscheiden sich bezüglich folgender Merkmale:

- *Wann* wird die Variable “erzeugt“ (Speicherplatz reserviert) und *wann* wird die Variable wieder “zerstört“ (zugehöriger Speicherbereich wieder freigegeben)?
- *Wo* ist die Variable *bekannt* (durch ihren Namen ansprechbar)?
- *Welchen Wert* bekommt die Variable implizit (ohne Initialisierung)?
- *Wie* muss eine explizite *Initialisierung* aussehen?
- *Wo* wird die Variable im Datenteil eines Programms *abgelegt*?

In der Einleitung wurde bereits erwähnt, dass der Datenteil eines Programms in mehrere Bereiche unterteilt ist (vgl. Abbildung 10 auf Seite 39).

In einem Teil, dem sog. *Stack*, werden die gewöhnlichen Variablen und Parameter der Funktionen abgelegt. Da beim Aufruf einer Funktion neue Variablen erzeugt werden und diese beim Funktionsende wieder zerstört werden, folgt, dass auf diesem Stack mal mehr, mal weniger Variablen abgelegt sind und dass sich die Variablenbelegung dieses Stacks dynamisch, während des Programmablaufs, ständig ändert.

Darüberhinaus gibt es andere Variablen (externe, statisch interne und statisch externe, s.u.), welche beim Programmstart ein für allemal erzeugt werden und bis zum Programmende bestehen bleiben. Diese Variablen werden in einem anderen Bereich des Datenteils (statischer Speicher) abgelegt.

Ein dritter Bereich, der sog. *Heap*, steht dem Programmierer eines Programms zur eigenen Speicherverwaltung mittels der Bibliotheksfunktionen `malloc` und `free` (vgl. Abschnitt 11.2) zur Verfügung.

9.1.1 Automatische Variablen

Bisher haben wir Funktionsparameter und innerhalb von Verbundanweisungen definierte Variablen kennengelernt.

So, wie wir es bislang kennengelernt haben innerhalb von Verbundanweisungen definierte Variablen sind sog. *automatische Variablen*.

Einer solchen Variablenvereinbarung kann (muss nicht) das Schlüsselwort `auto` vorangestellt werden. (Innerhalb von Verbundanweisungen definierte Variablen sind, wenn bei ihrer Definition nichts anderes angegeben ist, immer automatische Variablen! Ist der Typ der Variablen `int` und ist das Schlüsselwort `auto` angegeben, so kann der Typname `int` sogar fortgelassen werden.)

Merkmale einer solchen automatischen Variablen:

- Die Variable wird jedesmal, wenn der Programmfluss in die entsprechende Verbundanweisung eintritt (bzw. die Funktion aufgerufen wird), erneut erzeugt, und jedesmal am Ende der Verbundanweisung (bzw. Funktionsende) wieder zerstört.
- Bekannt und unter ihrem Namen ansprechbar ist die Variable nur innerhalb der Verbundanweisung (bzw. Funktion), in der sie definiert ist.
- Bei ihrer Erzeugung wird die Variable nicht implizit vorbesetzt, d.h. ist etwa wie in

```
int i;
```

keine explizite Initialisierung angegeben, so ist der Wert der Variablen zunächst zufällig.

- Ist bei der Variablendefinition, etwa wie in

```
int i = j * 2;
```

eine explizite Initialisierung angegeben, so kann der initialisierende Ausdruck (das, was hinter dem Gleichheitszeichen steht) ein beliebiger Ausdruck sein — kann also andere Variablennamen oder auch Funktionsaufrufe enthalten. Die explizite Initialisierung wird jedesmal bei der Erzeugung der Variablen durchgeführt, also bei jedem erneuten Beginn der entsprechenden Verbundanweisung bzw. Funktion erneut.

- Der Speicherbereich einer solchen Variablen wird auf einem dynamisch wachsenden und schrumpfenden *Stack* abgelegt:

Zunächst werden beim Programmstart die (automatischen) Variablen des Hauptprogramms `main` auf diesem Stack abgelegt.

Ruft das Hauptprogramm etwa eine andere Funktion (`f`) auf, so werden deren (automatische) Variablen auf dem Stack, hinter denen des Hauptprogramms, abgelegt. (D.h. der Stack wird länger!)

Wird diese Funktion (`f`) beendet, so werden deren auf dem Stack liegenden (automatischen) Variablen wieder gelöscht. (D.h. der Stack wird wieder kürzer.)

Ruft die Funktion (`f`) jedoch eine weitere Funktion (`g`) auf, so werden die (automatischen) Variablen von `g` auf dem Stack hinter denen von `f` abgelegt — erst wenn `g` beendet ist, werden die zugehörigen Variablen auf dem Stack gelöscht, so dass nur noch die von `f` und dem Hauptprogramm übrig sind.

Bei Beendigung von `f` werden die zugehörigen `f`-Variablen gelöscht und bei Beendigung des Hauptprogramms werden schließlich auch die (automatischen) Variablen des Hauptprogramms vom Stack entfernt.

Sinngemäß trifft dies auch für Verbundanweisungen anstelle der aufgerufenen Funktionen zu!

Ich möchte auf folgende Sonderfälle hinweisen:

1. Es kann mehrere gleichnamige Variablen geben, etwa in

```
void f(void)
{ int i;
  int x;
  ...
}
```

```

    for( i = 0; i < 100; ++i)
        { double x;
          ...
        }
    ...
}

```

D.h. zunächst wird in der Funktion `f` eine automatische Variable `int x` definiert und anschließend innerhalb der Verbundanweisung zur `for`-Schleife eine automatische Variable `double x`.

Wird nun die Schleifen-Verbundanweisung ausgeführt (dies geschieht hundertmal beim Ablauf der Funktion), so gibt es dann zwei unterschiedliche Speicherbereiche mit dem Namen `x` (einen vom Typ `double` und einen vom Typ `int`). Greift man innerhalb der Schleifen-Verbundanweisung auf den Namen `x` zu, so erhält man den `double`-Speicherbereich, greift man jedoch innerhalb der Funktion, aber außerhalb der Schleifen-Verbundanweisung auf den Namen `x` zu, so erhält man den `int`-Speicherbereich.

Die zugrundeliegende C-Regel ist:

Automatische Variablen "überdecken" gleichnamige "übergeordnete" Variablen!

2. Verwendet man eine rekursive Funktion, etwa

```

int f (int k)
{ int i;
  ...
  if ( k == 1)
    ...
  else
    ... f( k - 1 );
  ...
}

```

so wird beim ersten Aufruf dieser Funktion (u.a.) ein Speicherbereich für die `int`-Variable `i` (auf dem Stack) erzeugt. Bei diesem Funktionsaufruf wird aber die Funktion `f`, bevor der erste Aufruf abgeschlossen und somit der Speicherbereich wieder freigegeben werden konnte, erneut aufgerufen und hierbei erneut auf dem Stack ein weiterer Speicherbereich für die `int`-Variable `i` reserviert usw..

D.h. in der Rekursion existiert die Variable `i` mehrfach, einmal pro laufender Reinkarnation der rekursiven Funktion.

Funktionsparameter verhalten sich wie automatische Variablen, mit dem Unterschied, dass sie bei Ihrer Erzeugung automatisch einen wohldefinierten Wert erhalten, nämlich den Wert des korrespondierenden Funktionsargumentes beim Aufruf, und dass dementsprechend keine explizite Initialisierung möglich ist!

9.1.2 Register-Variablen

Bezüglich Verwendung und Merkmalen nahezu identisch zu den automatischen Variablen oder Funktionsparametern sind die sogenannten *Register-Variablen*. Sie werden wie automatische Variablen zu Beginn einer Verbundanweisung oder als Funktionsparameter mit dem vorhergehenden Schlüsselwort **register** (ohne das Schlüsselwort **auto**) definiert, also

```
int fkt( register long i,...) { ... }
```

oder

```
int f( ...)  
{ register i;  
  ...  
}
```

(Ist wie im letzten Beispiel das Schlüsselwort **register**, aber keine Typangabe angegeben, so wird implizit **int** als Typ angenommen!)

Auch diese register-Variablen

- werden bei jedem Aufruf der Funktion bzw. beim jedem Ablauf der entsprechenden Verbundanweisung erneut erzeugt und am entsprechenden Ende wieder zerstört,
- sind nur innerhalb der Funktion bzw. der Verbundanweisung bekannt,
- haben bei ihrer Erzeugung einen zufälligen Wert (falls sie nicht explizit initialisiert werden bzw. Funktionsparameter sind)
- werden, falls eine explizite Initialisierung (beliebiger initialisierender Ausdruck) vorliegt, bei jeder Erzeugung aufs neue mit dem entsprechenden Wert vorbesetzt (nicht bei Funktionsparametern!),
- überdecken gleichnamige übergeordnete Variablen.

Das einzige unterscheidende Merkmal ist, dass bei register-Variablen der Compiler *gebeten* wird, die entsprechende Variable in einem (besonders schnellen) Register (Speicherplatz auf dem Prozessor), und nicht im üblichen Variablen-Stack, abzulegen.

Wird diese Variable im betreffenden Programmteil sehr, sehr häufig verwendet, so kann ihre Unterbringung in einem CPU-Register Laufzeitvorteile mit sich bringen. (Programm wird schneller!)

Wie gesagt, handelt es sich um eine Bitte an den Compiler, er kann, muss dieser Bitte aber nicht nachkommen! Aufgrund des begrenzten Speicherbereiches einer CPU können nur einige wenige Variablen von “kleinen” Typen (z.B. keine Felder) in einem Register abgespeichert werden.

Unabhängig davon, ob die mittels **register** vereinbarte Variable vom Compiler nun tatsächlich in einem CPU-Register abgelegt worden ist oder wie jede sonstige (automatische) Variable auf dem Stack, kann auf diese Variable der Adress-Operator **&** nicht angewendet werden!

9.1.3 Statische interne Variablen

Statische interne Variablen werden innerhalb, zu Beginn einer Verbundanweisung (nicht als Funktionsparameter möglich) mit dem vorhergehenden Schlüsselwort **static** definiert. Etwa:

```
int f (...)  
{ static int i;  
    ...  
}
```

(auch hier wird bei fehlender Typangabe der Typ `int` angenommen). Statische interne Variablen haben folgende Merkmale:

- Statische interne Variablen werden beim Programmstart erzeugt und erst beim Programmende wieder zerstört. (Die Erzeugung und Zerstörung ist also nicht an Aufrufe der Funktion bzw. Verbundanweisung gebunden, in der sie definiert werden!)
- Bekannt und durch ihren Namen ansprechbar sind sie nur innerhalb der Funktion bzw. Verbundanweisung, in der sie definiert sind.
- Wird eine statische interne Variable nicht explizit initialisiert, so hat sie den Wert 0 (alle Bits sind 0!).
- Wird eine statische interne Variable explizit initialisiert, so muss der initialisierende Ausdruck ein *konstanter Ausdruck* sein, d.h. es dürfen keine Variablen oder Funktionsaufrufe beteiligt sein! (Somit steht der Wert des Ausdrucks zur Compilierzeit bereits fest!)
- Die Initialisierung (implizit mit 0 oder eine explizite) wird genau einmal beim Programmstart durchgeführt.
- Die Variable wird nicht im sich dynamisch ändernden Variablen-Stack angelegt, sondern in einem beim Programmstart vom System fest vereinbarten Speicherbereich (statischer Speicher).

Zur Verdeutlichung möchte ich folgendes Beispiel anführen:

```
int f (...)  
{ static int i = 7;  
    ...  
}
```

Für diese Variable wird beim Programmstart Speicher reserviert (nicht im Variablen-Stack) und dieser Speicherbereich wird mit dem ganzzahligen Wert 7 vorbesetzt. Anschließend beginnt das Programm und eventuell wird hierbei die Funktion *f* aufgerufen. Erst beim Ablauf dieser Funktion wird die Variable *i* “sichtbar“ (kann verwendet werden) und hat zunächst den Wert 7, mit dem sie vorbesetzt wurde. Beim Ablauf der Funktion *f* kann der Variablen *i* z.B. ein neuer Wert zugewiesen werden. Bei Beendigung der Funktion *f* bleibt die Variable *i* jetzt jedoch erhalten, wird aber “unsichtbar“ (kann also nicht verwendet werden). Erst beim nächsten Aufruf von *f* wird sie wieder “sichtbar“ und hat dann den Wert, den sie am Ende des letzten Aufrufs von *f* hatte. (Obwohl zu Beginn des Quelltextes von *f* die Zeile `static int i = 7;` steht, wird die Variable nicht neu erzeugt und nicht neu initialisiert — dies ist ein für allemal beim Programmstart bereits geschehen!)

9.1.4 Externe Variablen

Externe Variablen werden (wie Funktionen) außerhalb von Funktionen (außerhalb von geschweiften Klammern) definiert — üblicherweise vor der ersten Funktionsdefinition eines Quelltextes oder zwischen Funktionsdefinitionen. Etwa:

```
/* Quelltext A */
#include <stdio.h>

int i;

void main(void)
{ ... }

int j = 7;

int fkt1(int k)
{ ... }

void fkt2(void)
{ ... }
```

Externe Variablen haben folgende Merkmale:

- Wie eine statisch interne Variable wird eine externe Variable genau einmal beim Programm erzeugt und bleibt bis zum Programmende bestehen.
- Wird eine externe Variable (wie im obigen Beispiel die Variable `i`) bei ihrer Definition nicht explizit initialisiert, so erhält sie implizit bei ihrer Erzeugung den Wert 0 (alle Bits sind 0).
- Wird eine externe Variable (wie im obigen Beispiel die Variable `j`) bei ihrer Definition explizit initialisiert, so muss der initialisierende Ausdruck (wie bei statischen internen Variablen) ein *konstanter Ausdruck* sein (der Wert des Ausdrucks muss zur Compilierzeit bereits feststehen). Die entsprechende Initialisierung wird einmal beim Programmstart vorgenommen.
- Eine externe Variable wird beim Programmstart in demselben Speicherbereich angelegt, in dem auch statische interne Variablen angelegt werden (wiederum im statischen Speicher und nicht im Variablen-Stack!).
- Eine externe Variable ist unterhalb ihrer Definition im ganzen Quelltext bekannt. (D.h. im obigen Beispiel kann die Variable `i` in allen beteiligten Funktionen (`main`, `fkt1` und `fkt2`) verwendet werden und die Variable `j` nur in den letzten beiden Funktionen (`fkt1` und `fkt2`).)
- Darüberhinaus kann eine externe Variable “überall“ (natürlich nur zu Beginn einer Verbundanweisung oder wiederum außerhalb von Funktionsdefinitionen), wo sie benötigt wird, nochmals deklariert werden.

Die Deklaration einer externen Variablen sieht formal genauso aus wie ihre Definition, nur dass vorher das Schlüsselwort `extern` anzugeben ist.

```
extern int i;
```

Durch eine Deklaration wird der Typ und der Name der Variablen nochmals bekanntgemacht, ohne dass die Variable hierdurch (neu) definiert wird. Der Compiler erfährt dadurch von der Existenz der entsprechenden Variablen und er geht davon aus, dass sie an anderer Stelle irgendwo (als externe Variable) definiert ist. Das Auffinden dieser anderen Stelle ist dann Sache des Linkers und dieser gibt ggf. eine Fehlermeldung aus, wenn die entsprechende Variable überhaupt nirgendwo oder fälschlicherweise zweimal — in unterschiedlichen Quelltexten — definiert wurde!

Der Linker sorgt im lauffähigen Programm dafür, dass, wenn die Variable deklariert und sie verwendet wird, auf den möglicherweise in einem anderen Quelltext bei der Definition der Variablen reservierten Speicherbereich zugegriffen wird.

Erfolgt die Variablendeklaration innerhalb einer Verbundanweisung, so ist diese Variable nur innerhalb dieser Verbundanweisung bekannt und verwendbar.

Erfolgt die Variablendeklaration außerhalb von Funktionen, so ist die Variable wiederum von der Deklaration an bis zum Ende des Quelltextes bekannt und in jeder im Rest des Quelltextes definierten Funktion verwendbar.

Zur Verdeutlichung des letzten Punktes “Deklaration von externen Variablen“ soll unser oben bereits skizzierter Quelltext durch folgenden Quelltext ergänzt werden und beide Quelltexte zusammen sollen zu einem lauffähigen Programm gehören:

```
/* Quelltext B */

void fkt3(void)
{ /* Deklaration der externen Variablen j,
   Deklaration innerhalb von { ... } */
  extern int j;
  ...
}

/* Deklaration der externen Variablen i ,
   Deklaration ausserhalb von { ... } */
extern int i;

int fkt4(void)
{ ... }

void fkt5(int k)
{ /* Deklaration der externen Variablen j,
   Deklaration innerhalb von { ... } */
  extern int j;
  ...
}
```

Die Variable `i` ist im Quelltext B außerhalb einer Funktionsdefinition deklariert. Diese Deklaration macht die Variable im Rest des Quelltextes B bekannt — die Variable

kann hier also in den Funktionen `fkt4` und `fkt5` verwendet werden, nicht jedoch in Funktion `fkt3`, da deren Definition vor der Deklaration von `i` steht.

Die Variable `j` ist im Quelltext B jeweils innerhalb der Funktionen `fkt3` und `fkt5` deklariert — sie kann hier also in diesen Funktionen, nicht jedoch in Funktion `fkt4`, verwendet werden, da sie in Funktion `fkt4` nicht deklariert und somit unbekannt ist.

In allen zu einem lauffähigen Programm gehörenden Quelltexten muss eine externe Variable genau einmal definiert werden, sie kann mehrfach (auch innerhalb eines Quelltextes) deklariert werden. (Eine doppelte Deklaration stört nicht!)

Wird eine externe Variable in einem Quelltext (ziemlich weit unten) definiert, so kann sie auch vorher im selben Quelltext (etwa innerhalb einer im Quelltext weiter oben definierten Funktion) deklariert werden.

Typ und Name müssen bei Definition und Deklaration einer externen Variablen übereinstimmen. Es darf keine zwei unterschiedliche externe Objekte mit dem gleichen Namen geben. (Auch Funktionen sind extern, s.u. — eine externe Variable darf nicht den gleichen Namen wie eine, ggf. in einem anderen zum Programm gehörenden Quelltext definierte Funktion haben!)

Wird ein Feld als externe Variable definiert, so muss, wie üblich, bei der Definition eine konstante Feldlänge in eckigen Klammern angegeben werden. Bei der Deklaration kann auf die Angabe einer Feldlänge verzichtet werden — die (dann leeren) eckigen Klammern müssen jedoch bei der Deklaration aufgeführt sein!

Externe Variablen werden auch *globale* Variablen genannt. Es handelt sich um Variablen, auf die *globaler* Zugriff, also (falls entsprechend deklariert) von jeder Funktion des Programms aus, möglich ist.

Im Gegensatz hierzu sind automatische oder auch statisch interne Variablen oder auch register-Variablen *lokal*, d.h. auf diese kann nur *lokal* von der Funktion aus (bzw. innerhalb “ihrer” Verbundanweisung) zugegriffen werden. Außerhalb sind sie nicht bekannt.

Die Deklaration (nicht die Definition) externer Variablen kann man wiederum in eigene Header-Dateien schreiben und diese, überall wo notwendig, in den Quelltext includen. (Man kann natürlich in eine Header-Datei sowohl Funktions-Deklarationen als auch Vereinbarung von Aufzählungstypen als auch mittels `typedef` definierte “eigene” Typen als auch Deklarationen externer Variablen gemischt hineinschreiben!)

Hat eine automatische (oder statische interne oder eine register-) Variable einer Funktion (oder Verbundanweisung) den gleichen Namen wie eine externe Variable, so wird beim Ablauf der Funktion bzw. der Verbundanweisung die externe Variable “ausgeblendet“, d.h. innerhalb ist nur die lokale Variable “sichtbar“. Nach Beendigung der Funktion (bzw. Verbundanweisung) ist die externe Variable wieder “sichtbar“ (verwendbar).

9.1.5 Statische externe Variablen

Statische externe Variablen werden wie externe Variablen außerhalb von Funktionsdefinitionen, dann aber mit dem zusätzlichen Schlüsselwort `static`, definiert:

```
#include <stdio.h>

static int i;

void main(void)
{
    ...
}

int fkt1(int k)
{
    ...
}
```

Statische externe Variablen haben (bis auf eine Ausnahme) die gleichen Merkmale wie externe Variablen, also

- werden beim Programmstart genau einmal erzeugt und “leben“ bis zum Programmende,
- haben, wenn sie nicht explizit initialisiert werden, implizit den Wert 0,
- eine eventuelle explizite Initialisierung (mit einem dann notwendigerweise konstanten Ausdruck) wird einmal beim Programmstart durchgeführt,
- werden im gleichen Speicherbereich wie externe und statisch interne Variablen angelegt,
- können im Quelltext, in dem sie definiert sind, von ihrer Definition an in allen Funktionen des Quelltextes verwendet werden

Die Ausnahme ist, dass auf statisch externe Variablen nicht von anderen Quelltexten aus zugegriffen werden kann — sie können in anderen Quelltexten nicht deklariert werden — gleichwohl können sie im “eigenen“ Quelltext, also in dem sie definiert sind, etwa vor ihrer Definition deklariert werden! (Bei einem etwaigen Versuch, eine statische externe Variable in einem anderen Quelltext zu deklarieren und zu verwenden liefert der Linker eine entsprechende Fehlermeldung — ihm ist diese Variable unbekannt!)

Darüberhinaus können statisch externe Variablen den gleichen Namen haben wie externe Variablen oder Funktionen anderer zum Programm gehörender Quelltexte — im aktuellen Quelltext kann dann aber nur auf diese eigene, statische externe (Quelltext-interne) Variable und nicht auf die gleichnamige Variable oder Funktion des anderen Quelltextes zugegriffen werden.

Auf diese Weise hat man die Möglichkeit, eine in einem Quelltext notwendige globale Variable (als global notwendig, da etwa alle Funktionen dieses Quelltextes auf diese Variable zugreifen sollen) vor anderen Quelltexten zu “verstecken“. (Diese Variable kann in anderen Quelltexten eben nicht mehr deklariert werden!)

Darüberhinaus kann man einem Namen, welcher in allen anderen Quelltexten eine feste Bedeutung hat (etwa: Name einer externen `int` Variablen) in einem speziellen Quelltext eine andere Bedeutung zu geben (etwa: Name einer `double`-Variablen).

9.2 Speicherklassen von Funktionen

Alle bisher kennengelernten Funktionen sind extern und es gibt keine internen (etwa automatischen) Funktionen — eine Funktionsdefinition muss außerhalb jeder anderen Funktionsdefinition stattfinden.

Die bisher kennengelernten Funktionen entsprechen von ihrer Bekanntheit (Aufrufbarkeit) her der Bekanntheit (Verwendbarkeit) externer Variablen: Sie können unterhalb ihrer Definition im gleichen Quelltext überall aufgerufen werden und darüberhinaus überall dort, wo sie nochmals deklariert sind. (Da Funktionen per se extern sind, kann bei Funktionsdeklarationen das Schlüsselwort **extern** fortgelassen werden — es besteht ja auch im Gegensatz zu externen Variablen keine Verwechslungsgefahr zwischen Funktionsdefinition und Funktionsdeklaration.)

9.2.1 Statische Funktionen

Einer Funktionsdefinition kann das Schlüsselwort **static** vorangestellt werden:

```
static int fkt(void) {...}
```

Hierdurch wird die Bekanntheit und Verwendbarkeit (wie bei statischen externen Variablen) auf den aktuellen Quelltext eingeschränkt. D.h. diese Funktion kann innerhalb des aktuellen Quelltextes unterhalb ihrer Definition verwendet werden und sie kann in diesem Quelltext (etwa oberhalb ihrer Definition) wie gewöhnlich deklariert und anschließend verwendet werden — sie kann aber nicht in anderen Quelltexten deklariert und aufgerufen werden. (Bei einem etwaigen Versuch, diese in einem anderen Quelltext zu deklarieren und zu verwenden liefert der Linker eine entsprechende Fehlermeldung — ihm ist diese Funktion unbekannt!)

Eine statische Funktion kann den gleichen Namen haben wie eine in einem anderen Quelltext definierte Funktion oder externe Variable.

10 Der Präprozessor

Die Hauptanwendungen für den Präprozessor sind

- Einfügen von Dateien
- Ersetzen von Makros
- bedingte Übersetzung

Anweisungen an den Präprozessor sind Zeilen, welche mit dem „#“-Zeichen als erstem sichtbaren Zeichen beginnen (üblicherweise ist das „#“-Zeichen das erste Zeichen der Zeile überhaupt!).

Diese Zeile mit ihren eventuellen Folgezeilen („\ <cr> “ am Ende der vorherigen Zeile) richtet sich nur an den Präprozessor. Sie veranlasst den Präprozessor zu gewissen Aktionen (Textersetzungen) und verschwindet vor dem Übersetzen aus dem Quelltext.

10.1 Einfügen von Dateien

Zum Einfügen einer Datei stehen die beiden Präprozessoranweisungen

```
#include < Dateiname >      und  
#include " Dateiname "
```

zur Verfügung. In beiden Fällen wird die entsprechende Zeile durch den Inhalt der angegebenen Datei ersetzt.

Derartige einzufügende Dateien haben i. Allg. die Endung `.h` und heißen *Einfügedateien* oder *Header-Dateien* oder *Include-Dateien*.

In der ersten Form (mit spitzen Klammern) wird dort nach der einzufügenden Datei gesucht, wo üblicherweise im System die Einfügedateien (insbesondere die aus der Standardbibliothek) abgelegt sind.

In der zweiten Form (doppelte Anführungszeichen) wird die Datei zunächst im aktuellen Verzeichnis gesucht und, falls die Datei dort nicht gefunden werden konnte, in den sonst üblichen Verzeichnissen (wie bei der ersten Form mit den spitzen Klammern!). Wird die angegebene Datei nicht gefunden, so führt dies zu einem Übersetzungsfehler.

10.2 Ersetzen von Makros

Die am häufigsten verwendete Form von Makros ist:

```
#define Identifier Ersetzungstext
```

Jedesmal, wenn in der folgenden Quelldatei *Identifier* als „eigenständiges“ Wort (*TOKEN*, kein Teilwort, nicht innerhalb doppelter Anführungszeichen) auftritt, wird *Identifier* durch *Ersetzungstext* ersetzt.

Im Ersetzungstext kann man „Folgezeilen“ definieren, indem man am Ende der Zeile den Zeilenvorschub durch ein „\“ maskiert“. (D.h. der Ersetzungstext besteht im Quelltext aus mehreren Zeilen, im ersetzten Text jedoch nur aus einer, da der maskierte Zeilenvorschub vom Compiler ignoriert wird!)

Symbolische Konstanten sind typische Beispiele für derartige Makros.

C bietet die Möglichkeit, Makros mit Argumenten zu definieren und mit diesen die Textersetzung variabler zu gestalten.

Eine entsprechende Definition lautet:

```
#define Identifer( Parameterliste ) Ersetzungstext
```

Hierbei darf zwischen *Identifer* und der öffnenden Klammer kein Zwischenraum sein! *Parameterliste* ist eine Liste von Namen.

Tritt nun innerhalb des folgenden Quelltextes der Name des Makros (*Identifer*) auf, hinter dem (nach eventuellen Zwischenraumzeichen) eine runde Klammer auf, eine Liste von Argumenten und eine runde Klammer zu steht, so wird *Identifer* zusammen mit Klammern und Argumentliste durch den Ersetzungstext ersetzt. (Die Anzahl der Einträge der Parameterliste und der Argumentliste müssen hierbei übereinstimmen!)

Tritt im Ersetzungstext ein Name aus der Parameterliste als eigenständiges Wort (Token) auf, so wird dieser Name durch das ihm entsprechende Argument des „Makroaufrufs“ ersetzt.

Beispiel:

Aufgrund der Definition

```
#define MAX(a,b)    ( (a) > (b) ) ? (a) : (b)
```

wird im folgenden Quelltext

```
t = MAX(r,s);      durch
```

```
t = ( (r) > (s) ) ? (r) : (s);      ersetzt.
```

(Die Zuweisung und der Strichpunkt gehören nicht zum Makro, diese stehen im Quelltext vor bzw. hinter dem Makro und werden entsprechend unverändert übernommen!)

Dieses Makro mit Argumenten bestimmt also typunabhängig das Maximum zweier Größen, für die der Vergleichsoperator > definiert ist.

Bei Makros mit Argumenten muss man sorgfältig auf korrekte Klammerung achten, wie man in folgendem Makro zum (typunabhängigen) Quadrieren von Werten sieht:

```
#define QUADRAT(a)    a * a
```

Ruft man das Makro beispielsweise wie folgt auf:

```
QUADRAT( x + 1 )
```

so wird dieser Aufruf vom Präprozessor wie folgt ersetzt:

```
x + 1 * x + 1
```

wobei aufgrund der „Rechenregeln“ der Wert $2 \cdot x + 1$ und nicht wie erhofft das Quadrat von $x + 1$ berechnet wird.

Die Definition des Makros müsste richtig:

```
#define QUADRAT(a)    ((a) * (a))
```

lauten.

Bei der Verwendung von Makros mit Argumenten bedürfen Seiteneffekte besonderer Aufmerksamkeit, etwa:

```
MAX( r++ , s++ );
```

wird zu

```
( (r++) > (s++) ) ? (r++) : (s++);
```

expandiert.

Der größere der beiden Werte `r` und `s` wird somit zweimal inkrementiert. Dies sieht man dem „Aufruf“ des Makros nicht an!

Makros mit Argumenten kann man (mit „kleinen“ Unterschieden) wie Funktionen verwenden. Da die Ersetzung aber vor dem Compilieren erfolgt, wird der interne Verwaltungsaufwand für Funktionsaufrufe beim Programmlauf vermieden.

Einige „Funktionen“ der Standardbibliothek sind als Makros mit Argumenten realisiert!

Die Zeichen „#“ vor und „##“ vor oder hinter einem Parameter im Ersetzungstext verursachen eine gesonderte Textersetzung:

steht „#“ vor einem Parameter im Ersetzungstext, so wird „#Parameter“ durch das in doppelte Anführungszeichen gestellte aktuelle Argument ersetzt. Sollten im aktuellen Argument selbst doppelte Anführungszeichen („“) oder ein Backslash („\“) vorkommen, so werden diese durch „\“ bzw. durch „\\“ ersetzt.

Beispiel:

Durch die Präprozessoranweisung

```
#define PRINTD(A)    printf( #A " = %d \n",A)
```

wird nun in dem folgenden Programm etwa

```
PRINTD(x);           durch
printf( "x" " = %d \n", x);    ersetzt.
```

Durch die implizite Verkettung adjazenter Zeichenketten (siehe Seite 43) wird hieraus

```
printf("x = %d \n", x);
```

Durch diese Makro-Konstruktion kann man also Namen von `int`-Variablen oder auch `int`-Ausdrücke als Zeichenketten und deren Werte gleichzeitig ausgeben, etwa:

```
int x = 2, y = 3;
PRINTD(x*y + 2);
PRINTD(7 + 5);
```

wird vom Präprozessor ersetzt durch:

```
int x = 2, y = 3;
printf("x*y + 2" " = %d \n", x*y + 2);
printf("7 + 5" " = %d \n", 7 + 5);
```

und beim Programmablauf erscheint folgende Ausgabe auf dem Bildschirm:

```
x*y + 2 = 8
7 + 5 = 13
```

Ein „##“ im Ersetzungstext wird inklusive benachbarten Zwischenraumzeichen (Blanks, Tab's) bei der Ersetzung fortgelassen. So können beispielsweise zwei Identifier zu einem neuen Identifier aneinandergekettet werden.

Ersetzungen werden gegebenenfalls mehrfach durchgeführt:

Durch

```
#define cat( x, y)    x ## y
#define MAXWORT      128
```

wird im folgenden Text

```
cat ( MAX, WORT)      zunächst durch
MAXWORT              ersetzt und anschliessend durch
128.
```

Durch

```
#undef Identifier
```

kann man erreichen, dass im folgenden Quelltext *Identifier* nicht ersetzt wird, falls er vorher durch ein **#define** als Makro (ohne oder mit Argumente) definiert wurde. Die Makrodefinition wird also für den Rest der Quelldatei ausgeblendet.

10.3 Bedingte Übersetzung

Es gibt Auswahlanweisungen für den Präprozessor, welche, je nachdem ob eine Bedingung *falsch* oder *wahr* ist, die folgenden Zeilen vor der Übersetzung aus dem Quelltext entfernt oder nicht.

Die möglichen Formen sind:

```
#if Bedingung
    Text    (mehrere Zeilen)
#endif
```

Wenn die *Bedingung falsch* ist, wird *Text* vor der Übersetzung entfernt.

```
#if Bedingung
    Text1    (mehrere Zeilen)
#else
    Text2    (mehrere Zeilen)
#endif
```

Wenn die *Bedingung wahr* ist, wird *Text2* entfernt, sonst *Text1*.

Schachtelungen sind mit **#elif** möglich:

```
#if Bedingung1
:
#elif Bedingung2
:
#elif Bedingung3
:
#else
:
#endif
```

(Der **#else**-Teil ist hierbei optional!)

Als *Bedingung* sind jeweils ganzzahlige konstante Ausdrücke zugelassen, welche insbesondere keinen `sizeof`-Operator, keine Umwandlungen und keine Aufzählungskonstanten enthalten.

Der Ausdruck wird vom Präprozessor ausgewertet und als *wahr* interpretiert, wenn das ganzzahlige Resultat ungleich 0 ist.

Steht in der *Bedingung* ein Ausdruck der Form

```
defined Identifier      bzw.
```

```
defined (Identifier)
```

so wird dieser Ausdruck durch 1 ersetzt, falls *Identifier* der Name eines definierten Makros (mit oder ohne Argumente) ist und 0 sonst.

Die Präprozessorzeilen

```
#ifdef Identifier      bzw.
```

```
#ifndef Identifier
```

sind gleichbedeutend zu

```
#if defined Identifier      bzw.
```

```
#if ! defined Identifier.
```

Hierdurch wird folgende Technik möglich, das mehrfache (ggf. auch indirekte) Einbinden ein- und derselben Header-Datei in ein- und denselben Quelltext zu verhindern (das mehrfache Einbinden sollte zwar an sich nicht schädlich sein, wenn in der Header-Datei etwa nur Deklarationen stehen, trotzdem wird der Quelltext für den Compiler unnötigerweise aufgebläht!):

```
#ifndef NAME
#define NAME
.
. /* eigentlicher Inhalt der Header-Datei */
.
#endif
```

In der zweiten Zeile der Header-Datei wird ein Makro (hier **NAME**) definiert (auch ohne Ersetzungstext möglich) und in der ersten Zeile, also vorher, mit **#ifndef** gefragt, ob dieses nicht definiert ist.

Beim erstmaligen Einbinden der Header-Datei ist dieses Makro noch nicht definiert und die **#ifndef**-Abfrage somit wahr und der eigentliche Inhalt der Header-Datei wird eingebunden — u.a. auch die Definition des Makros.

Bei einem nochmaligen Einbinden ist dieses Makro dann ja bereits definiert, die **#ifndef**-Abfrage somit falsch und die zum wiederholten male eingebundene Header-Datei wird durch den Präprozessor selbst wieder entfernt!

10.4 Vordefinierte Makros

Folgende Namen sind vom Präprozessor vordefiniert und dürfen nicht undefiniert werden:

<code>__LINE__</code>	Ganzzahlige Konstante, welche die Zeilennummer der aktuellen Zeile enthält.
<code>__FILE__</code>	Konstante Zeichenkette, welche den Namen der gerade übersetzten Quelldatei enthält.
<code>__DATE__</code>	Konstante Zeichenkette, die das Datum der Übersetzung enthält.
<code>__TIME__</code>	Konstante Zeichenkette, die die Uhrzeit der Übersetzung enthält.
<code>__STDC__</code>	Konstante, die von der Implementierung nur dann definiert werden darf, wenn sie dem Standard genügt.

10.5 Weitere Präprozessoranweisungen

Die Präprozessoranweisung

`#line` Konstante "Name" bzw.

`#line` Konstante

setzt den Wert von `__LINE__` auf den Wert der angegebenen Konstante und bei gegebenem Namen wird `__FILE__` auf diesen Namen gesetzt.

Die Präprozessoranweisung

`#error` Fehlertext

bringt den Präprozessor dazu, (u.a.) die angegebene Fehlermeldung auszugeben und den Compilervorgang abubrechen.

Angewendet werden kann diese Anweisung etwa wie folgt:

```
#ifndef UNIX
    /* UNIX-spezifische Vereinbarungen durchfuehren */
    ...
#elif defined MSDOS
    /* MS-DOS-spezifische Vereinbarungen durchfuehren */
    ...
#else
    #error Kein Systemtyp bekannt!
#endif
...
```

Hier wird davon ausgegangen, dass der Systemtyp (UNIX oder MSDOS) vorher als Makro (evtl. in einer systemeigenen, vorher eingebundenen Header-Datei) definiert wurde. Ist das System ein UNIX-System (und das entsprechende Makro definiert), werden in die UNIX-spezifischen Vereinbarungen alle auf UNIX-Systemen erforderlichen Einstellungen vorgenommen, der MS-DOS-Teil und die Fehler-Präprozessoranweisung werden entfernt und anschließend wird der Compiler aufgerufen.

Ist der Systemtyp MS-DOS (und das entsprechende Makro definiert), so werden nur die entsprechenden MS-DOS-spezifischen Einstellungen vorgenommen und anschließend wird kompiliert.

Ist jedoch keiner der beiden Systemtypen (als Makro) definiert, so landet der Präprozessor in dem **#else**-Teil, in dem die **#error**-Anweisung steht. Hierdurch gibt der Präprozessor den Dateinamen (Inhalt des **__FILE__**-Makros), die Zeilennummer (Inhalt des **__LINE__**-Makros) und die angegebene Fehlermeldung

Kein Systemtyp bekannt!

aus und der Übersetzungsvorgang wird beendet.

Die Wirkung der Präprozessoranweisung

#pragma ...

ist implementierungsabhängig.

Die leere Präprozessoranweisung

#

wird effekt- und ersatzlos aus dem Quelltext entfernt.

11 Die Standardbibliothek

Der eigentliche Sprachumfang von C ist im Vergleich zu anderen höheren Programmiersprachen relativ klein. Dies liegt daran, dass einige, insbesondere die hardware-spezifischen Sprachelemente, welche man i. Allg. von einer höheren Programmiersprache erwartet, in C nicht Bestandteil des eigentlichen Kerns der Sprache sind, sondern durch eine standardisierte Bibliothek von Funktionen zur Verfügung gestellt werden. Der Standard definiert die Eigenschaften (Name, Rückgabetyt und Parametertypen) dieser Bibliotheksfunktionen sehr genau und die meisten Implementierungen (alle, die sich ANSI-compatibel nennen) realisieren sie so, dass sie sich wie im Standard beschrieben verhalten.

Die Details der jeweiligen Realisierung sind natürlich maschinen- und implementationsabhängig.

Zu diesen durch Bibliotheksfunktionen realisierten Bereichen gehören u.a.:

- Ein- und Ausgabe
- Dateizugriff
- Speicherverwaltung
- mathematische Funktionen

Es folgt nach Sachgebieten geordnet eine kurze Beschreibung aller Funktionen der ANSI-Standardbibliothek.

11.1 Standardein/ausgabe

Die Funktionen, Makros und Typen zur Ein/Ausgabe sind in der Header-Datei `stdio.h` deklariert. Sie unterstützen ein einfaches Modell für Ziel und Quelle von Daten, nämlich den sogenannten Datenstrom. Ein Datenstrom ist eine geordnete Folge von einzelnen Zeichen (Bytes). Das Ende des Datenstroms wird nicht als Zeichen des Stroms aufgefasst.

Man unterscheidet den Binärstrom (binärer Datenstrom) vom Textstrom. Beim Binärstrom sind alle Zeichen gleichberechtigt; beim Textstrom haben gewisse Zeichen, insbesondere das Zeilenendezeichen, eine Sonderbedeutung. (Textströme setzen sich aus Zeilen zusammen, jede Zeile besteht aus keinem, einem oder mehreren Zeichen und (einem) anschließendem Zeilenendezeichen. Manche Systeme stellen ein Zeilenende durch mehrere Zeichen dar. Es ist u.a. Aufgabe der Bibliotheksfunktionen, die physikalisch wie auch immer vorhandenen Dateien in dieser Modellform dem C-Programm zugänglich zu machen.)

Derartige Datenströme sind (oder werden) mit Bildschirm, Tastatur, Sekundärspeicher oder anderer Peripherie verknüpft.

Standardmäßig sind die drei Textströme Standardeingabe, Standardausgabe und Standardfehlerausgabe in einem C-Programm bekannt. Die Standardeingabe ist normalerweise mit der Tastatur und die Standardausgabe sowie Standardfehlerausgabe sind gewöhnlich mit dem Bildschirm verknüpft.

Einige Betriebssysteme ermöglichen, diese Ein-/Ausgabeströme umzulenken (vgl. Abschnitt 1.2.6).

11.1.1 Elementare Ein-/Ausgabe

Die wichtigsten Funktionen zur (formatierten) Ein-/Ausgabe mit diesen Kanälen werden im folgenden aufgeführt:

```
int getchar(void);
```

Die Funktion `getchar` liefert bei jedem Aufruf das nächste Zeichen der Standardeingabe (genauer: es wird der Wert des „gelesenen“ Zeichens als `int`-Wert als Funktionsergebnis zurückgegeben.). Das Zeilenendezeichen zählt hierbei auch als (ein) Zeichen. Wird das Ende des Eingabestroms erreicht, so liefert `getchar` den Wert `EOF`, der als symbolische Konstante in `stdio.h` (meistens als `-1`) definiert ist. (`EOF` darf nicht der Wert eines Zeichens des Zeichensatzes sein.)

```
int putchar(int);
```

`putchar(c)` gibt das Zeichen, dessen Wert durch den ganzzahligen Ausdruck `c` gegeben ist, auf dem Standardausgabekanal (also auf dem Bildschirm) aus. Die Funktion liefert als Funktionsergebnis den Wert des ausgegebenen Zeichens zurück oder `EOF`, falls ein Fehler aufgetreten ist.

```
char *gets(char *s);
```

`gets` liest die nächste Zeile (einschließlich `'\n'`) von der Standardeingabe, legt das Gelesene in der Zeichenkette `s` ab und ersetzt in `s` das Zeilenendezeichen `'\n'` durch das Stringendezeichen `'\0'`. Funktionsergebnis ist `s` oder `NULL` im Fehlerfall.

```
int puts(const char *s);
```

`puts` schreibt die Zeichenkette `s` und ein anschließendes Zeilenendezeichen `'\n'` auf die Standardausgabe. Das Funktionsergebnis ist größer 0 oder `EOF` im Fehlerfall.

```
int printf(const char *format,...);
```

Die Funktion `printf` dient zur Ausgabe interner Werte (Werte von Variablen und von Ausdrücken) als Zeichenfolgen. An ihrer Deklaration sieht man, dass es sich um eine Funktion mit variabler Argumentliste handelt (vgl. Abschnitt 11.3).

Der Formatstring (die Zeichenkette, auf die der Zeiger `format` zeigt!) wird zusammen mit den unter Kontrolle des Formatstrings in Zeichenfolgen umgewandelten weiteren Argumenten auf dem Bildschirm ausgegeben.

Das Funktionsergebnis der Funktion `printf` ist die (positive) Anzahl der ausgegebenen Zeichen bzw. negativ im Fehlerfall.

Der Formatstring enthält gewöhnliche Zeichen (Zeichen, die nicht zwischen einem „%“ und dem nächsten Umwandlungszeichen, s.u., stehen), die auf die Ausgabe kopiert werden, und Umwandlungsspezifikationen.

Jede Umwandlungsspezifikation beginnt mit dem „%“-Zeichen und endet mit einem Umwandlungszeichen.

Jede Umwandlungsspezifikation (Ausnahme: „%%“) bezieht sich auf ein weiteres Argu-

ment des Funktionsaufrufs von `printf` (die erste Umwandlungsspezifikation im Formatstring bezieht sich auf das erste zusätzliche Argument, die zweite auf das zweite zusätzliche Argument etc.).

Die Umwandlungszeichen und ihre Bedeutung sind in folgender Tabelle aufgeführt:

Zeichen	Argument, Umwandlung in
d, i	<code>int</code> ; dezimal mit Vorzeichen (<code>signed</code>).
o	<code>int</code> ; oktal ohne Vorzeichen (ohne führende Null).
x, X	<code>int</code> ; hexadezimal ohne Vorzeichen (ohne führendes 0x bzw. 0X), mit <code>abcdef</code> als zusätzliche Ziffern bei <code>ox</code> und <code>ABCDEF</code> bei <code>0X</code> .
u	<code>int</code> ; dezimal ohne Vorzeichen (<code>unsigned</code>).
c	<code>int</code> ; einzelnes Zeichen nach Umwandlung in <code>unsigned char</code> .
s	<code>char *</code> ; aus der Zeichenkette, auf die der Zeiger zeigt, werden die Zeichen bis zum nächsten <code>'\0'</code> (ausschließlich), höchstens aber so viele, wie die Präzision angibt, ausgegeben.
f	<code>double</code> ; dezimal als <code>[-]mmm.d...d</code> , wobei die Präzision die Anzahl der <code>d</code> 's festlegt. Voreinstellung ist 6; bei 0 entfällt der Dezimalpunkt.
e, E	<code>double</code> ; dezimal als <code>[-]m.d...de±xx</code> bzw. <code>[-]m.d...dE±xx</code> , wobei wiederum die Präzision die Anzahl der <code>d</code> 's festlegt. Voreinstellung ist 6; bei 0 entfällt der Dezimalpunkt.
g, G	<code>double</code> ; wenn der Exponent kleiner als <code>-4</code> ist oder nicht kleiner als die Präzision, wird <code>%e</code> bzw. <code>%E</code> verwendet, sonst <code>%f</code> . Nullen und Dezimalpunkt am Schluss werden nicht ausgegeben.
p	<code>void *</code> ; implementierungsabhängige Ausgabe von Zeigerwerten.
n	<code>int *</code> ; hierbei wird kein Argument umgewandelt und ausgegeben, sondern es wird die Anzahl der bisher von diesem <code>printf</code> ausgegebenen Zeichen in die <code>int</code> -Variable geschrieben, auf die der Zeiger verweist.
%	es wird kein Argument umgewandelt sondern ein „%“-Zeichen ausgegeben.

Zwischen dem „%“-Zeichen und dem Umwandlungszeichen kann in einer Ausgabespezifikation der Reihe nach folgendes angegeben werden:

- Steuerzeichen in beliebiger Reihenfolge. Steuerzeichen modifizieren die Umwandlung.
 - bewirkt, dass das umgewandelte Argument linksbündig ausgegeben wird.
 - + bewirkt, dass Zahlen immer mit Vorzeichen ausgegeben werden.
- Leerzeichen* bewirkt, dass, wenn das erste Zeichen einer umgewandelten Zahl kein Vorzeichen ist, ein Leerzeichen ausgegeben wird.
- 0 bewirkt, dass bei numerischen Größen bis zur Feldbreite (s.u.) mit führenden Nullen aufgefüllt wird.
- # legt je nach Umwandlungszeichen eine gesonderte Form der Ausgabe fest:
 - bei `o` als Umwandlungszeichen wird als erste Ziffer eine Null ausgegeben.
 - bei `x` bzw. `X` werden `0x` bzw. `0X` vor einem von Null verschiedenen Wert

geschrieben.

- bei **e**, **E**, **f**, **g** und **G** enthält die Ausgabe immer einen Dezimalpunkt; bei **g** und **G** werden darüberhinaus Nullen am Schluss nicht unterdrückt.
- Eine Zahl, welche die Mindestfeldbreite angibt: das umgewandelte Argument wird mit mindestens soviel Zeichen, wie die Feldbreite angibt, ausgegeben (bei Bedarf mit mehr!). Hat das umgewandelte Argument weniger als in der Feldbreite angegebenen Zeichen, so wird mit Leerzeichen (oder mit Nullen, falls das entsprechende Steuerzeichen vorliegt) nach links (oder nach rechts, wenn das entsprechende Steuerzeichen vorliegt) aufgefüllt.
- Ein Punkt, der die Feldbreite von der Präzision trennt.
- Eine Zahl, die die Präzision angibt. Der Effekt der Präzision richtet sich nach dem Umwandlungszeichen.
 - Von Zeichenketten (Umwandlungszeichen **s**) werden maximal so viele Zeichen ausgegeben, wie die Präzision angibt.
 - Bei Gleitkommagrößen und **e**, **E** oder **f** als Umwandlungszeichen gibt die Präzision die Anzahl der auszugebenden Ziffern hinter dem Dezimalpunkt an.
 - Bei Gleitkommagrößen und **g** oder **G** als Umwandlungszeichen gibt die Präzision die Anzahl der signifikanten Ziffern an.
 - Werden ganzzahlige Werte ausgegeben, so gibt die Präzision die minimale Anzahl von Ziffern an, die ausgegeben werden (bis zur gewünschten Breite werden dann führende Nullen hinzugefügt!).
- Eine Längenangabe in Form eines der Buchstaben **h**, **l** oder **L**.
 - **h** bedeutet, dass das zugehörige Argument **short** oder **unsigned short** ist.
 - **l** bedeutet, dass das zugehörige Argument **long** oder **unsigned long** ist.
 - **L** bedeutet, dass das zugehörige Argument **long double** ist.

Als Feldbreite oder Präzision kann jeweils auch „*“ angegeben werden. In diesem Fall wird der tatsächliche Wert der Feldbreite bzw. Präzision (oder von beiden) durch das nächste Argument (oder von den nächsten beiden Argumenten) von **printf** festgelegt. (Dieses Argument oder diese Argumente müssen dann den Typ **int** besitzen!)

```
int scanf(const char *format,...);
```

Die zu **printf** duale Funktion **scanf** dient zum „Einlesen“ von Daten. **scanf** liest unter Kontrolle des Formatstrings **format** Zeichen von der Standardeingabe, wandelt diese nach den Umwandlungsspezifikationen im Formatstring in Werte um und legt diese Werte in den weiteren Argumenten des Aufrufs von **scanf** ab. (Wegen des Prinzips „Call by Value“ müssen diese Argumente natürlich vom Zeigertyp sein!) Das Funktionsergebnis von **scanf** ist die Anzahl der umgewandelten und abgelegten Eingaben bzw. EOF, falls das Dateiende erreicht worden oder ein Fehler aufgetreten ist.

Der Inhalt einer Formatzeichenkette kann aus folgenden Arten von Zeichen bestehen:

- Leerzeichen und Tabulatorzeichen. Diese werden ignoriert.

- Gewöhnliche Zeichen (Zeichen, die nicht zwischen einem „%“-Zeichen und dem nächsten Umwandlungszeichen stehen (s.u.)). Es wird erwartet, dass das nächste sichtbare Zeichen des Eingabestroms diesem Zeichen entspricht.
- Umwandlungsangaben. Diese bestehen aus dem „%“-Zeichen, dem optionalem Zeichen „*“, einer optionalen natürlichen Zahl (Feldbreite), bei gewissen Umwandlungszeichen (s.u.) die optionalen Zeichen „h“, „l“ oder „L“ und dem (nicht optionalen) Umwandlungszeichen.

Eine Umwandlungsangabe (Eingabespezifikation) legt die Umwandlung des nächsten Eingabefeldes fest.

Ein Eingabefeld ist eine Folge von sichtbaren Zeichen, die keine Zwischenraumzeichen sind (Zwischenraumzeichen sind Leerzeichen, Tabulatorzeichen `'\t'`, Zeilentrenner `'\n'`, Wagenrücklauf `'\r'`, vertikaler Tabulator `'\v'` und Seitenvorschub `'\f'`).

Falls keine explizite Feldbreite (natürliche Zahl) in der Umwandlungsangabe angegeben wurde, erstreckt sich das Eingabefeld bis zum nächsten Zwischenraumzeichen. (Sonst hat das Eingabefeld die Feldbreite als Länge!)

Das „*“-Zeichen in der Umwandlungsangabe unterdrückt eine Zuweisung; das entsprechende Eingabefeld wird gelesen, jedoch nicht umgewandelt und zugewiesen.

Ist das entsprechende Argument ein Zeiger auf `short` statt auf `int`, so ist den Umwandlungszeichen `d`, `i`, `n`, `o`, `u` bzw. `x` ein `h` voranzustellen.

Entsprechend ist diesen Umwandlungszeichen ein `l` voranzustellen, wenn es sich beim Argument um einen Zeiger auf `long` handelt.

Ein vorangestelltes `l` bei den Umwandlungszeichen `e`, `f` oder `g` bedeutet, dass der entsprechende Zeiger ein Zeiger auf `double` anstelle von `float` ist.

Ein vorangestelltes `L` bei den Umwandlungszeichen `e`, `f` oder `g` bedeutet, dass der entsprechende Zeiger ein Zeiger auf `long double` anstelle von `float` ist.

Die Umwandlungszeichen sind in folgender Tabelle aufgeführt:

Zeichen	Eingabedaten, Argumenttyp
<code>d</code>	dezimal, ganzzahlig; <code>int *</code> .
<code>i</code>	ganzzahlig; <code>int *</code> . Der Wert kann oktal (mit 0 am Anfang) oder hexadezimal (mit <code>0x</code> oder <code>0X</code> am Anfang) angegeben sein.
<code>o</code>	oktal ganzzahlig (mit oder ohne 0 am Anfang); <code>int *</code> .
<code>x</code>	hexadezimal ganzzahlig (mit oder ohne <code>0x</code> oder <code>0X</code> am Anfang); <code>int *</code> .
<code>c</code>	ein oder mehrere Zeichen; <code>char *</code> . Die nachfolgenden Eingabezeichen werden im angegebenen Vektor abgelegt, bis die Feldbreite erreicht ist; Voreinstellung ist 1. <code>'\0'</code> wird nicht angefügt. In diesem Fall wird Zwischenraum nicht überlesen; das nächste Zeichen nach Zwischenraum liest man mit „%1s“ (wobei aber zwei Zeichen, das eingelesene und das <code>'\0'</code> -Zeichen, abgelegt werden!).
<code>s</code>	Folge von Nicht-Zwischenraum-Zeichen (ohne Anführungszeichen); <code>char *</code> , der auf einen Vektor zeigen muss, der die Zeichenkette und das abschließende <code>'\0'</code> -Zeichen aufnehmen kann.

Zeichen	Eingabedaten, Argumenttyp
e, f, g	Gleitpunkt; <code>float *</code> . Das Eingabeformat erlaubt für <code>float</code> ein optionales Vorzeichen, eine Ziffernfolge, die auch einen Dezimalpunkt enthalten kann, und einen optionalen Exponenten, bestehend aus <code>e</code> oder <code>E</code> und einer ganzen Zahl, optional mit Vorzeichen.
p	Zeiger, wie ihn <code>printf("%p")</code> ausgibt; <code>void *</code> .
n	legt im Argument die Anzahl der Zeichen ab, die bei diesem Aufruf bisher gelesen wurden; <code>int *</code> . Vom Eingabestrom wird nicht gelesen, die Zählung der Umwandlungen bleibt unbeeinflusst.
[...]	erkennt die längste nicht-leere Zeichenkette aus den Eingabezeichen in der angegebenen Klasse; <code>char *</code> . Dazu kommt <code>'\0'</code> . Die Klasse <code>[]...</code> enthält auch <code>]</code> .
[^...]	erkennt die längste nicht-leere Zeichenkette aus den Eingabezeichen, die <i>nicht</i> in der angegebenen Klasse sind; <code>char *</code> . Dazu kommt <code>'\0'</code> . Die Klasse <code>[^]...</code> enthält auch <code>]</code> .
%	erkennt %; eine Zuweisung findet nicht statt (kein zugehöriges Argument!).

11.1.2 Dateizugriff

Ein Textstrom oder auch ein Binärstrom kann mittels der Bibliotheksfunktion `fopen` mit einer Datei des Systems verbunden werden.

Diese Funktion liefert als Ergebnis einen Zeiger auf ein Objekt vom (in `stdio.h` definierten) Typ `FILE` (oder den Wert `NULL` im Fehlerfall).

In diesem Objekt verbirgt sich vor dem Benutzer die für den Zugriff auf Dateien (fürs Betriebssystem) notwendige Information. Der Benutzer benötigt hingegen nur die Adresse dieses Objektes, die er in einen Zeiger auf den Typ `FILE` ablegen sollte.

Die genaue Deklaration der Funktion `fopen` lautet:

```
FILE *fopen(const char *filename, const char *mode);
```

`filename` ist der Name der Datei, die als Strom geöffnet werden soll und

`mode` gibt an, in welchem Modus die Datei geöffnet werden soll.

Dateinamen dürfen nicht länger sein als die systemabhängige Größe `FILENAME_MAX` und es dürfen gleichzeitig nicht mehr Dateien geöffnet sein als `FOPEN_MAX`.

Die grundlegenden Modi sind:

- "r" Die Datei wird zum Lesen (read) geöffnet. Existiert die angegebene Datei nicht, so gibt `fopen` den Wert `NULL` zurück.
- "w" Die Datei wird zum Schreiben (write) geöffnet. Existiert die Datei noch nicht, so wird sie angelegt. Existiert die Datei bereits, so wird deren Inhalt gelöscht.
- "a" Die Datei wird zum Schreiben (append) geöffnet. Existiert die Datei noch nicht, so wird sie angelegt. Existiert die Datei bereits, so wird an ihrem Ende weitergeschrieben.

Den Zeichen `r`, `w` bzw. `a` kann das Zeichen „+“ oder das Zeichen „b“ (oder auch beide) hinzugefügt werden.

Der Modus "r+" bedeutet, dass auf die angegebene Datei unter gewissen Einschränkungen auch geschrieben werden kann (Aktualisieren von Daten).

Die Modi "w+" und "a+" bewirken entsprechend, dass von der Datei unter gewissen Einschränkungen auch gelesen werden kann.

Das „b“-Zeichen in der Modusangabe zeigt an, dass es sich um eine Datei handelt, die als Binärstrom angesehen wird.

Ein nicht erfolgreicher Öffnungsversuch einer Datei liefert als Funktionsergebnis den Wert NULL. (Üblicherweise kontrolliert man in C-Programmen, ob das Öffnen einer Datei erfolgreich war, um ggf. das Programm entsprechend reagieren zu lassen — etwa: Fehlermeldung und Abbruch.)

Geöffnete Dateien werden mittels der Funktion

```
int fclose(FILE *stream);
```

geschlossen. Hierbei wird beim Schreiben ein eventueller Pufferinhalt (siehe Abschnitt 11.1.3) noch auf die Datei geschrieben, die Datei geschlossen und der Filepointer **stream** kann für eine weitere Datei verwendet werden.

Alle Dateien werden beim Programmende (nicht aber beim Programmabsturz!) automatisch geschlossen.

Standardmäßig sind die drei „Filepointer“ (Zeiger vom Typ FILE) **stdin** (Standardeingabe), **stdout** (Standardausgabe) und **stderr** (Standardfehlerausgabe) in jedem Programm bekannt. Wie bereits erwähnt sind gewöhnlich **stdin** mit der Tastatur sowie **stdout** und **stderr** mit dem Bildschirm verknüpft.

stderr ist üblicherweise nicht gepuffert und eignet sich somit besonders zur Fehlerausgabe.

Die Funktion

```
FILE *freopen( const char *filename, const char *mode, FILE *stream);
```

funktioniert ähnlich wie **fopen**. Das Funktionsergebnis wird auch im Parameter **stream** abgelegt. Mit **freopen** kann man die Ströme **stdin**, **stdout** und **stderr** umlenken.

Formatierte Ein/Ausgabe von/auf mittels Filepointer zugänglichen „Dateien“ geschieht mit den folgenden Bibliotheksfunktionen, die den unter Standardein/ausgabe beschriebenen Funktionen entsprechen mit der Ergänzung, dass der Filepointer als weiteres Argument angegeben werden muss!

- `int fputc(int c, FILE *stream);`
Funktioniert wie `putc`.
- `int putc(int c, FILE *stream);`
Wie `putc`, kann aber Makro sein!
- `int fputs(const char *s, FILE *stream);`
Wie `puts`.
- `int fprintf(FILE *stream, const char *format,...);`
Wie `printf`.
- `int fgetc(FILE *stream);`
Wie `getchar`.
- `int getc(FILE *stream);`
Wie `getchar`, kann aber Makro sein!

- `char *fgets(char *s, int n, FILE *stream);`
Wie `gets`, liest aber höchstens $n - 1$ Zeichen.
- `int fscanf(FILE *stream, const char *format,...);`
Wie `scanf`.

Beispiel:

```
#include <stdio.h>

void main(void)
{
    FILE *op;

    if ((op = fopen("ausgabedatei","w")) == NULL)
        fprintf(stderr," Ausgabedatei kann nicht beschreiben werden\n");
    else
        fprintf(op," hello, world\n");
    fclose(op);
}
```

In diesem Beispielprogramm wird versucht, eine Datei mit dem Namen „ausgabedatei“ zum Schreiben zu öffnen und die Zeichenkette "hello, world\n" auf sie zu schreiben. Falls das Öffnen nicht möglich ist (falls man etwa nicht das Recht hat, eine Datei zu erzeugen!), wird eine Fehlermeldung auf den Standardfehlerkanal geschrieben.

Die Funktion

```
int ungetc(int c, FILE *stream);
```

stellt das (in unsigned char umgewandelte) Zeichen mit dem Wert `c` auf den Eingabestrom `stream` zurück, so dass es beim nächsten Lesen zuerst gelesen wird. Funktionsergebnis ist das „zurückgelesene“ Zeichen oder EOF im Fehlerfall.

Von `printf/fprintf` bzw. `scanf/fscanf` gibt es die folgenden Abwandlungen:

```
int sprintf(char *s, const char *format,...);
```

funktioniert wie `printf`, es wird jedoch nicht auf einen Ausgabestrom, sondern in die Zeichenkette `s` geschrieben, wobei am Ende ein Stringendezeichen '\0' angehängt wird, welches im Funktionsergebnis (Anzahl der geschriebenen Zeichen) jedoch nicht mitgezählt wird. Der vorherige Inhalt der Zeichenkette `s` wird gegebenenfalls überschrieben! Ein Überlaufen der Zeichenkette wird nicht überprüft.

```
int sscanf(char *s, const char *format,...);
```

funktioniert wie `scanf`, es wird jedoch nicht von einem Eingabestrom, sondern aus der Zeichenkette `s` gelesen.

Die Funktionen

```
int vprintf(const char *format, va_list arg);
```

```
int vfprintf(FILE *const char *format, va_list arg);
```

```
int vsprintf(char *s, const char *format, va_list arg);
```

sind äquivalent zu den entsprechenden `printf`-Funktionen mit dem Unterschied, dass die variable Argumentliste ... durch ein Argument vom Typ `va_list` ersetzt wird (vgl. Abschnitt 11.3). Die `v?printf`-Funktionen können etwa innerhalb einer Funktion aufgerufen werden, die selbst ihre Argumente über eine variable Argumentliste erhält. Die variable Argumentliste der aufgerufenen Funktion kann dann an die entsprechende `v?printf`-Funktion weitergereicht werden.

11.1.3 Weitere Dateioperationen

Wie bereits erwähnt sind Ausgabeströme üblicherweise gepuffert, d.h. Ausgaben werden intern zwischengespeichert und erst dann ausgegeben, wenn der Zwischenspeicher (meist 512 oder 1024 Byte) voll ist. Beim ordnungsgemäßen Programmende wird der Pufferinhalt ebenfalls ausgegeben. Sinnvollerweise wird vor Eingaben aus `stdin` der Puffer für `stdout` ebenfalls ausgegeben.

Die Funktion

```
int fflush( FILE *stream);
```

sorgt dafür, dass die für den Ausgabestrom `stream` gepufferte, noch nicht geschriebene Ausgabe herausgeschrieben wird. Das Funktionsergebnis ist 0 nach fehlerfreiem Ablauf bzw. EOF sonst. Der Aufruf `fflush(NULL)` bezieht sich auf alle offenen Ausgabeströme.

Die Pufferung von Ausgabeströmen kann man mit den Funktionen

```
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

```
void setbuf(FILE *stream, char *buf);
```

beeinflussen (`size_t` ist hierbei der Datentyp, der vom `sizeof`-Operator geliefert wird, vgl. Abschnitt 8.1). Diese Funktionen müssen nach dem Öffnen des Ausgabestromes, vor jeder anderen Aktion auf dem Strom, aufgerufen werden. Das Argument `stream` ist jeweils der Ausgabestrom, dessen Pufferung beeinflusst werden soll.

Beim Aufruf von `setvbuf` gibt `size` die Größe des Puffers an und als `mode` kann einer der folgenden Werte angegeben werden:

- `_IONBF` Es wird nicht gepuffert.
- `_IOLBF` Es wird gepuffert und jedesmal bei vollem Puffer, spätestens jedoch nach jedem Zeilenendezeichen `'\n'` und beim Schließen des Ausgabestromes wird der Pufferinhalt herausgeschrieben.
- `_IOFBF` Es wird gepuffert und jedesmal bei vollem Puffer, spätestens beim Schließen des Ausgabestromes wird der Pufferinhalt herausgeschrieben.

Ist das Argument `buf` gleich `NULL`, so wird vom System ein interner Puffer angelegt. Ansonsten kann `buf` die Adresse eines Zeichenfeldes mit mindestens der Länge `size` sein, welches dann als Puffer verwendet wird. Im Fehlerfall liefert die Funktion den Wert 0, sonst einen von 0 verschiedenen Wert.

Hat beim Aufruf der Funktion `setbuf` der Parameter `buf` den Wert `NULL`, so wird nicht gepuffert. Ansonsten ist der Aufruf gleichbedeutend zu

```
(void) setvbuf(stream, buf, _IOFBF, BUFSIZ),
```

wobei `BUFSIZ` eine implementationsabhängige Standardgröße für Puffer ist.

Die Funktion

```
int remove(const char *filename);
```

löscht die angegebene Datei und liefert den Wert 0, falls hierbei kein Fehler auftrat.

Die Funktion

```
int rename(const char *oldname, const char *newname);
```

ändert den Namen einer Datei und gibt beim fehlerfreien Verlauf den Wert 0 zurück.

Die Funktion

```
FILE *tmpfile(void);
```

erzeugt eine temporäre Datei, öffnet diese im Modus `"wb+"` und gibt einen Zeiger auf diese Datei zurück bzw. `NULL` im Fehlerfall. Die Datei wird automatisch beim Schließen der Datei gelöscht.

Beim Aufruf der Funktion

```
char *tmpnam( char *s);
```

wird ein Name erzeugt, der nicht Name einer existierenden Datei im aktuellen Verzeichnis ist.

Ist das Argument `s` gleich `NULL`, so wird dieser neue Name intern statisch abgelegt und die Adresse dieses Namens als Funktionswert zurückgegeben.

Ist `s` ungleich `NULL`, so muss `s` die Adresse eines Zeichenfeldes mit mindestens der implementationsabhängigen Länge `L_tmpnam` sein. Der neue Name wird in `s` abgelegt und `s` wird als Funktionsergebnis zurückgegeben.

Im Fehlerfall wird jeweils `NULL` geliefert.

Es können in einem Programmlauf höchstens `TMP_MAX` verschiedene Namen erzeugt werden.

11.1.4 Fehlerbehandlung bei Datenströmen

Ein Datenstrom kann sich in unterschiedlichen Zuständen befinden, u.a.

- Datenstrom ist in Ordnung
- das Ende des Datenstroms ist erreicht, der Datenstrom aber prinzipiell in Ordnung
- der Datenstrom ist nicht mehr in Ordnung (Fehler)

Der Zustand eines Datenstroms wird intern zum Datenstrom notiert. Bei einigen Fehlern wird die globale Variable `errno`, die in der Header-Datei `errno.h` deklariert ist, auf einen entsprechenden Wert gesetzt.

Ob das Ende des Datenstroms erreicht ist, kann man außer an den Funktionsergebnissen der Lesefunktionen mittels folgender Funktion feststellen:

```
int feof(FILE *stream);
```

liefert ein von 0 verschiedenes Ergebnis, wenn für den angegebenen Strom das Dateiende notiert ist.

Die Funktion

```
int ferror(FILE *stream);
```

liefert einen von 0 verschiedenen Wert, wenn für den angegebenen Datenstrom ein Fehler notiert ist.

Die Funktion:

```
void perror( const char *s);
```

gibt die angegebene Zeichenkette `s` und eine vom System vorgegebene Fehlermeldung, die sich auf die in `errno` abgelegte Fehlernummer bezieht, auf dem Standardfehlerkanal aus.

Die Funktion:

```
void clearerr(FILE *stream);
```

löscht eine Dateiende- bzw. eine Fehlernotierung für den angegebenen Datenstrom.

11.1.5 Direkte Ein-/Ausgabe, Dateipositionierung

Zur Positionierung eines Dateizeigers stehen die folgenden Funktionen zur Verfügung:

```
int fseek(FILE *stream, long offset, int origin);
```

setzt den Dateizeiger so, dass nachfolgende Lese- oder Schreiboperationen von der neuen Position ab zugreifen.

Bei Binärdateien wird die Position auf die Stelle `origin` plus `offset` in der Datei gesetzt. Der Parameter `origin` kann hierbei `SEEK_SET` für Dateianfang, `SEEK_END` für Dateiende bzw. `SEEK_CUR` für die aktuelle Dateiposition sein. Das `long`-Argument `offset` ist eine positive oder negative Anzahl von Bytes.

Textdateien können mittels `fseek` nur auf den Dateianfang: `fseek(stream, 0L, SEEK_SET)`; oder auf's Dateiende: `fseek(stream, 0L, SEEK_END)`; oder auf eine mittels `ftell` vorher ermittelten und abgespeicherten Position gesetzt werden (hierbei muss `origin` stets `SEEK_SET` sein!).

Das Ergebnis von `fseek` ist im Fehlerfall ungleich 0.

```
long ftell(FILE *stream);
```

liefert als Funktionsergebnis die aktuelle Dateiposition bzgl. des Dateianfangs bzw. `-1L` im Fehlerfall.

Wird eine Datei, deren Ende erreicht wurde und für die somit das Dateiende notiert ist, mittels `fseek` auf eine zurückliegende Position (etwa den Dateianfang) positioniert, so wird die Dateiendenotierung nicht automatisch gelöscht — dies geschieht erst durch den expliziten Aufruf von `clearerr` (siehe vorherigen Unterabschnitt).

Die Funktion

```
void rewind(FILE *stream);
```

positioniert den Dateizeiger auf den Dateianfang und löscht eine eventuelle Notiz, dass das Dateiende vorher bereits erreicht wurde.

```
int fgetpos(FILE *stream, fpos_t *ptr);
```

speichert die aktuelle Position in einem Objekt des implementationsabhängigen, zur Aufnahme derartiger Positionierungsdaten geeigneten Types `fpos_t`. Im Fehlerfall liefert `fgetpos` einen von 0 verschiedenen Wert.

```
int fsetpos(FILE *stream, fpos_t *ptr);
```

positioniert den Datenstrom `stream` auf die vorher mittels `fgetpos` ermittelte und in `*ptr` abgespeicherte Position. Im Fehlerfall liefert `fsetpos` einen von 0 verschiedenen Wert.

Direkte Ausgabe geschieht mit der Funktion

```
size_t fwrite(const char *ptr, size_t size, size_t nobj, FILE *stream);,
```

welche `nobj` Objekte der Größe `size` aus dem Vektor `ptr` auf den Datenstrom `stream` schreibt. Der Funktionswert ist die Anzahl der geschriebenen Objekte, im Fehlerfall also weniger als `nobj`.

Die Funktion

```
size_t fread(void *ptr, size_t size, size_t nobj, FILE *stream);
```

liest vom Datenstrom `stream` höchstens `nobj` der Größe `size` in den Vektor `ptr`. Funktionsergebnis ist die Anzahl der gelesenen Objekte, also im Fehlerfall weniger als `nobj`.

11.2 Universelle Hilfsfunktionen

In der Deklarationsdatei `stdlib.h` sind allgemein nützliche Hilfsfunktionen deklariert. Diese Hilfsfunktionen umfassen die (dynamische) Speicherverwaltung, Umwandlung von Zahlen in Zeichenketten und umgekehrt, Erzeugung von Pseudo-Zufallszahlen und universelle Such- und Sortierfunktionen.

Zur dynamischen Speicherverwaltung stehen folgende Funktionen zur Verfügung:

```
void *calloc(size_t nobj, size_t size);
```

Diese Funktion liefert einen `void`-Zeiger auf einen freien Speicherbereich für einen Vektor von `nobj` Objekten jeweils der Größe `size`. Dieser Speicherbereich ist mit Nullbytes initialisiert. Wenn die Anforderung nicht erfüllt werden konnte, wird `NULL` als Funktionsergebnis zurückgegeben.

Der Benutzer muss mittels des `cast`-Operators den Zeiger auf diesen Speicherbereich in den gewünschten Typ umwandeln (etwa `(int *) calloc(..., ...)`, falls der Speicherbereich zur Speicherung von `int`-Werten benötigt wird!).

```
void *malloc(size_t size);
```

liefert einen Zeiger auf einen Speicherbereich der Größe `size` oder `NULL`, falls die Anforderung nicht erfüllt werden konnte. Der Bereich ist nicht initialisiert.

```
void *realloc(void *p, size_t size);
```

ändert die Größe des Objektes, auf das der Zeiger `p` zeigt und welches vorher dynamisch angefordert wurde, auf `size` ab. Wird das Objekt größer, so ist der zusätzliche Bereich nicht initialisiert. Funktionsergebnis ist die Adresse des neuen Speicherbereiches. Kann die Anforderung nicht erfüllt werden, so wird `NULL` als Funktionswert geliefert und das ursprüngliche Objekt bleibt unverändert.

```
void free(void *p);
```

gibt den Bereich frei, auf den der Zeiger `p` verweist. Dieser Bereich muss mittels `calloc`, `malloc` oder `realloc` zuvor angefordert worden sein.

Hat `p` den Wert `NULL`, so bewirkt die Funktion `free` nichts.

Weitere in `stdlib.h` deklarierte Funktionen:

```
void exit(int status);
```

Der Aufruf dieser Funktion (auch aus Unterprogrammen heraus) bewirkt eine sofortige Beendigung des Programms. Der `int`-Wert `status` wird an das Betriebssystem als Ergebnis zurückgegeben.

```
void abort(void);
```

veranlasst eine sofortige, abnormale Beendigung des Programms.

```
int atexit(void (*fcn)(void));
```

Der Aufruf von `atexit` bewirkt, dass bei normalem Programmende die Funktion aufgerufen wird, auf die der Funktionszeiger `fcn` (siehe Abschnitt 13.2) zeigt. Diese Funktion muss folgenden Typ haben: keine Argumente und kein Funktionsergebnis. Bei einem fehlerhaften Ablauf von `atexit` wird ein von 0 verschiedenes Funktionsergebnis geliefert.

```
int system(const char *s);
```

Diese Funktion leitet die Zeichenkette, auf die der Zeiger `s` verweist, an das Betriebssystem zur Ausführung weiter. (D.h. in dieser Zeichenkette kann ein Systembefehl oder auch der Befehl zum Aufruf eines anderen Programms stehen.) Das Betriebssystem führt den so übermittelten Befehl aus und anschließend wird im laufenden C-Programm weiter fortgefahren. Der Resultatwert des Aufrufs von `system` ist von der Implementierung abhängig.

```
char *getenv(const char * name);
```

Diese implementationsabhängige Funktion ermöglicht es, Daten aus der aktuellen Benutzerumgebung (vielleicht Suchpfad o.ä.) in einem Programm verfügbar zu machen. Einige Implementierungen stellen den Zugriff auf die Benutzerumgebung mittels eines weiteren Funktionsargumentes zum Hauptprogramm `main` zur Verfügung.

```
double atof(const char *s);
```

wandelt die Zeichenkette `s` in `double` um.

```
int atoi(const char *s);
```

wandelt die Zeichenkette `s` in `int` um.

```
long atol(const char *s);
```

wandelt die Zeichenkette `s` in `long` um.

```
double strtod(const char *s, char **endp);
```

wandelt den Anfang der Zeichenkette `s`, der als `double` interpretiert werden kann, in `double` um und gibt einen Zeiger auf den nicht umgewandelten Rest der Zeichenkette im Parameter `endp` zurück, falls `endp` nicht `NULL` ist. Bei Überlauf (Overflow) ist das Funktionsergebnis der symbolische Wert `HUGE_VAL` (mit korrektem Vorzeichen), bei Unterlauf (underflow) wird der `double`-Wert 0 zurückgegeben. In beiden Fällen erhält

die globale Fehlervariable `errno` (in `errno.h` deklariert, vgl. Unterabschnitt 11.1.4) den Wert `ERANGE`.

```
long strtol(const char *s, char **endp, int base);
```

funktioniert ähnlich, nur dass ein `long`-Wert zurückgegeben wird.

Dem Parameter `base` kommt hierbei folgende Bedeutung zu:

Ist `base` gleich 0, so wird (der Anfang von) `s` im Dezimalsystem interpretiert bzw. bei führendem Zeichen 0 im Oktalsystem bzw. bei führendem `0x` oder `0X` im Hexadezimalsystem.

Sonst kann `base` jeden Wert von 2 bis 36 annehmen und die Zeichen in `s` werden im Zahlensystem zu Basis `base` interpretiert. Ggf. werden die Buchstaben von `A` beginnend als Ziffern in diesem Zahlensystem angesehen.

Wird die umgewandelte Zahl zu groß, wird `LONG_MAX` als Ergebnis geliefert und wird sie zu klein, wird `LONG_MIN` geliefert. In beiden Fällen erhält die globale Fehlervariable `errno` den Wert `ERANGE`.

```
unsigned long strtoul(const char *s, char **endp, int base);
```

funktioniert wie `strtol`, nur dass das Ergebnis vom Typ `unsigned long` ist und dass die umgewandelte Zahl höchstens zu groß werden kann. In diesem Fall wird `ULONG_MAX` als Funktionsergebnis geliefert.

```
int rand(void);
```

liefert eine ganzzahlige Pseudo-Zufallszahl im Bereich von 0 bis `RAND_MAX`; der Wert von `RAND_MAX` ist mindestens 32767.

```
void srand(unsigned int seed);
```

benutzt das Argument `seed` als Ausgangswert für eine (dann mittels der Funktion `rand` erzeugte) neue Folge von Pseudo-Zufallszahlen. Der erste Ausgangswert ist 1.

```
int abs(int n);
```

liefert den Absolutwert seines `int`-Argumentes.

```
long labs(long n);
```

liefert den Absolutwert seines `long`-Argumentes.

```
div_t div(int num, int denom);
```

berechnet Quotient und Rest der ganzzahligen Division von `num` durch `denom` und legt diese in den `int`-Komponenten `quot` und `rem` einer Struktur vom Typ `div_t` ab.

(Vom Standard durch

```
typedef {  
    int quot;  
    int rem;  
} div_t;
```

oder ähnlich definiert. Zu Strukturen siehe Abschnitt 12 !)

```
ldiv_t ldiv(long num, long denom);
```

berechnet Quotient und Rest der ganzzahligen Division von `num` durch `denom` und legt

diese in den `long`-Komponenten `quot` und `rem` einer (zu `div_t` analogen) Struktur vom Typ `ldiv_t` ab.

Funktion zur Binären Suche:

```
void *bsearch(const void *key, const void *base,
              size_t n, size_t size,
              int (*cmp)(const void *keyval, const void * datum));
```

`bsearch` durchsucht `base[0], ..., base[n-1]` nach einem Eintrag, der gleich `*key` ist. (Jeder Eintrag des Feldes `base` hat hierbei eine Länge von `size` Bytes.) Die Elemente des Feldes müssen aufsteigend sortiert sein! Die Ordnung, welche diese aufsteigende Reihenfolge beschreibt, wird durch die Funktion, auf die der Funktionszeiger (siehe Abschnitt 13.2) `cmp` verweist, festgelegt: die Vergleichsfunktion (`*cmp`) muss einen negativen Wert liefern, wenn ihr erstes Argument (`*keyval`) kleiner als ihr zweites Argument (`*datum`) ist, Null, wenn beide gleich sind, und sonst einen positiven Wert. Das Funktionsergebnis ist ein Zeiger auf das gefundene Element oder `NULL`, falls die Suche erfolglos war.

Sortieralgorithmus:

```
void qsort(void *base, size_t n, size_t size,
           int (*cmp)(const void *, const void *));
```

`qsort` sortiert einen Vektor `base[0], ..., base[n-1]` von `n` Objekten der Größe `size` in aufsteigender Reihenfolge. Die Ordnung, welche dem Sortieren zugrundeliegt, wird wiederum durch die Funktion `*cmp` festgelegt, deren Eigenschaften wie bei `bsearch` beschrieben sein müssen.

11.3 Variable Argumentliste

Durch die Makros in `stdarg.h` bietet der C-Standard die Möglichkeit, Funktionen mit Parameterlisten zu definieren, deren Länge und Datentypen von Aufruf zu Aufruf unterschiedlich sein können.

Eine derartige Funktion muss mindestens einen benannten Parameter und eine mit „,...)“ endende Parameterliste besitzen.

In einer solchen Funktion muss man dann eine Variable vom Typ `va_list` (dies ist ein in `stdarg.h` definierter Zeigertyp) definieren.

Die entsprechende Variable kann dann der Reihe nach auf jedes Argument der Argumentliste des tatsächlichen Aufrufs der Funktion zeigen.

Diese Variable ist mit dem Makro

```
va_start(argument-pointer, lastargument);
```

zu initialisieren, wobei `argument-pointer` der Name der Variablen vom Typ `va_list` und `lastargument` der Name des letzten benannten Parameters der Funktion ist.

Jeder nun folgende Aufruf des Makros

```
va_arg(argument_pointer, type);
```

liefert als Resultat den in den angegebenen Typ *type* umgewandelten Wert des jeweils nächsten unbenannten Argumentes des Funktionsaufrufes.

Nachdem alle Argumente abgearbeitet worden sind, muss schließlich vor dem Funktionsende das Makro

```
va_end(argument_pointer);
```

aufgerufen werden, damit eventuell notwendige Aufräumarbeiten erledigt werden. Aus dem letzten benannten Argument *lastargument* sollte die Länge der tatsächlichen Argumentliste beim Funktionsaufruf ersichtlich sein!

Im folgenden elementaren Beispiel wird eine Funktion

```
int maximum(int n,...);
```

definiert, welche mit `maximum(n, a1, ..., an)` aufgerufen das Maximum der *n* natürlichen Zahlen *a*₁, ..., *a*_{*n*} als Ergebnis liefert. *n* ist hierbei variabel, so dass z.B.

`maximum(2,a,b)`, `maximum(4,6,x,y,x-y)` und auch `maximum(7,1,2,3,4,5,6,7)` legale Aufrufe dieser Funktion sind!

```
#include <stdarg.h>
```

```
int maximum(int n,...)
```

```
{
```

```
    va_list ap;
```

```
    int ival, max = 0;
```

```
    int i;
```

```
    va_start(ap,n);
```

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

```
        { ival = va_arg(ap, int);
```

```
          if (ival > max) max = ival;
```

```
        }
```

```
    va_end(ap);
```

```
    return max;
```

```
}
```

11.4 Mathematische Funktionen

In der Header-Datei `math.h` sind die mathematischen Funktionen (und einige zugehörige Makros) deklariert. Auf UNIX-Rechnern muss zusätzlich beim Compilieren (genauer: beim Binden) die Option `-lm` angegeben werden — die Bibliothek (`libm.a`) mit den übersetzten mathematischen Funktionen wird standardmäßig nämlich nicht

hinzugebunden!

Bei trigonometrischen Funktionen sind Winkelangaben im Bogenmaß.

`double sin(double x);`

Sinusfunktion.

`double cos(double x);`

Cosinusfunktion.

`double tan(double x);`

Tangensfunktion.

`double asin(double x);`

Arcussinusfunktion. Ergebnis im Intervall $[-\frac{\pi}{2}, +\frac{\pi}{2}]$, Argument x im Intervall $[-1, 1]$.

`double acos(double x);`

Arcuscosinusfunktion. Ergebnis im Intervall $[0, \pi]$, Argument x im Intervall $[-1, 1]$.

`double atan(double x);`

Arcustangensfunktion. Ergebnis im Intervall $[-\frac{\pi}{2}, +\frac{\pi}{2}]$.

`double atan2(double x, double y);`

Arcustangens von $\frac{y}{x}$. Ergebnis im Intervall $[-\pi, \pi]$!

Das Ergebnis ist der Winkel zwischen der positiven x -Achse und dem Ortsvektor zum Punkt mit den Koordinaten (x, y) in einem cartesischen Koordinatensystem. Der Punkt $(0, 0)$ ist nicht zugelassen!

`double sinh(double x);`

Sinus-Hyperbolicus-Funktion.

`double cosh(double x);`

Cosinus-Hyperbolicus-Funktion.

`double tanh(double x);`

Tangens-Hyperbolicus-Funktion.

`double exp(double x);`

Exponentialfunktion.

`double log(double x);`

Natürlicher Logarithmus.

`double log10(double x);`

Logarithmus zur Basis 10.

`double pow(double x, double y);`

Potenzfunktion; es wird x^y berechnet. Ein Argumentfehler liegt vor bei $x = 0$ und $y \leq 0$ oder bei $x < 0$ und y nicht ganzzahlig.

`double sqrt(double x);`

Quadratwurzel. x muss größer oder gleich 0 sein.

```
double ceil(double x);
```

Kleinsten ganzzahligen Wert, der nicht kleiner als x ist. Ergebnis ist vom Typ `double`!

```
double floor(double x);
```

Größten ganzzahligen Wert, der nicht größer als x ist. Ergebnis ist vom Typ `double`!

```
double fabs(double x);
```

Absolutbetrag von x .

```
double ldexp(double x, int n);
```

Berechnet $x \cdot 2^n$.

```
double frexp(double x, int *exp);
```

Zerlegt das Argument x in $x = a \cdot 2^n$ mit $a \in [\frac{1}{2}, 1)$ und einer ganzen Zahl n . a wird als Funktionsergebnis zurückgegeben und n wird in `*exp` abgelegt.

```
double modf(double x, double *ip);
```

Zerlegt die Zahl x in die Summe von einem ganzzahligen Teil und einem Rest in $(-1, 1)$, welche beide das gleiche Vorzeichen wie x haben. Der ganzzahlige Teil wird in `*ip` (als `double`) abgelegt, der Rest wird als Funktionsergebnis zurückgegeben.

```
double fmod(double x, double y);
```

Gleitpunktrest ($x - (x/y) * y$) von x/y . Wenn x/y nicht dargestellt werden kann (z.B. wenn y gleich Null ist), hängt das Resultat von der Implementierung ab.

11.5 Tests für Zeichen

Die in `ctype.h` deklarierten Funktionen dienen zum Testen von Zeichen. Die Argumente der Funktionen müssen `int`-Werte sein, die entweder gleich `EOF` sind oder die als `unsigned char` dargestellt werden können.

Alle Funktionen haben `int` als Rückgabotyp und liefern ein von Null verschiedenes Ergebnis, wenn das Argument die getestete Bedingung erfüllt.

```
int isupper(int c);
```

Wahr, falls c ein Großbuchstabe (keine Umlaute!) ist.

```
int islower(int c);
```

Wahr, falls c ein Kleinbuchstabe (keine Umlaute!) ist.

```
int isdigit(int c);
```

Wahr, falls c eine Ziffer ist.

```
int iscntrl(int c);
```

Wahr, falls c ein Steuerzeichen ist.

```
int isalpha(int c);
```

Wahr, falls `isupper(c)` oder `islower(c)` wahr ist.

```
int isalnum(int c);
```

Wahr, falls `isalpha(c)` oder `isdigit(c)` wahr ist.

```
int isgraph(int c);
```

Wahr, falls `c` ein druckbares Zeichen (mit Ausnahme des Leerzeichens) ist.

```
int isprint(int c);
```

Wahr, falls `c` ein druckbares Zeichen (einschließlich Leerzeichen) ist.

```
int ispunct(int c);
```

Wahr, falls `c` ein druckbares Zeichen, jedoch kein Leerzeichen, kein Buchstabe und keine Ziffer ist.

```
int isspace(int c);
```

Wahr, falls `c` ein Zwischenraumzeichen (Leerzeichen, Seitenvorschub, Zeilentrenner, Wagenrücklauf, Tabulatorzeichen, Vertikaltabulator) ist.

```
int isxdigit(int c);
```

Wahr, falls `c` eine hexadezimale Ziffer ist.

Zusätzlich gibt es zwei Funktionen zur Umwandlung zwischen Klein- und Großbuchstaben:

```
int tolower(int c);
```

Falls `c` ein Großbuchstabe ist, wird der entsprechende Kleinbuchstabe zurückgegeben. Ansonsten wird `c` selbst zurückgegeben.

```
int toupper(int c);
```

Analog zu `tolower`.

11.6 Funktionen für Zeichenketten

Die Funktionen für Zeichenketten sind in der Header-Datei `string.h` deklariert.

```
char *strcpy(char *s, const char *t);
```

Die Zeichenkette `t` wird (inklusive terminierendem `'\0'`) nach `s` kopiert. Das Funktionsergebnis ist `s`.

```
char *strncpy(char *s, const char *t, size_t n);
```

Höchstens die ersten `n` Zeichen aus `t` werden nach `s` kopiert. Hat `t` weniger als `n` Zeichen, so wird `s` mit `'\0'`-Zeichen bis zur Länge `n` aufgefüllt. Funktionsergebnis ist wiederum `s`.

```
char *strcat(char *s, const char *t);
```

Die Zeichenkette `t` wird hinten an die Zeichenkette `s` angehängt. Hierbei wird das terminierende `'\0'` von `s` überschrieben. Das Funktionsergebnis ist `s`.

```
char *strncat(char *s, const char *t, size_t n);
```

Höchstens die ersten `n` Zeichen aus `t` werden an `s` angehängt. Ergebnis ist `s`.

```
int strcmp(const char *s, const char *t);
```

Zeichenketten `s` und `t` werden lexikographisch verglichen; Ergebnis kleiner als Null, falls `s` kleiner als `t` ist, Ergebnis gleich Null, falls `s` und `t` gleich sind, sonst Ergebnis größer als Null.

```
int strncmp(const char *s, const char *t, size_t n);
```

Höchstens die ersten `n` Zeichen der Zeichenketten `s` und `t` werden verglichen. Ergebnis des Vergleichs wie bei `strcmp`.

```
char *strchr(const char *s, int c);
```

Liefert einen Zeiger auf das erste in `s` auftretende Zeichen mit Wert `c` oder Null, falls kein solches Zeichen vorhanden ist.

```
char *strrchr(const char *s, int c);
```

Liefert einen Zeiger auf das letzte in `s` auftretende Zeichen mit Wert `c` oder Null, falls kein solches Zeichen vorhanden ist.

```
size_t strspn(const char *s, const char *t);
```

Liefert die Anzahl der Zeichen am Anfang von `s`, die sämtlich in `t` vorkommen.

```
size_t strcspn(const char *s, const char *t);
```

Liefert die Anzahl der Zeichen am Anfang von `s`, die sämtlich nicht in `t` vorkommen.

```
char *strpbrk(const char *s, const char *t);
```

Liefert einen Zeiger auf die Position in `s`, an der irgendein Zeichen aus `t` erstmals vorkommt oder NULL, falls keins vorkommt.

```
char *strstr(const char *s, const char *t);
```

Liefert Zeiger auf die erste Kopie des Strings `t` im String `s` oder NULL, falls nicht vorhanden.

```
size_t strlen(const char *s);
```

Liefert die Anzahl der Zeichen des Strings `s` (ohne abschließendes `'\0'`-Zeichen!).

```
char *strerror(size_t n);
```

Liefert Zeiger auf die Zeichenkette, die in der Implementierung für den Fehler mit der Nummer `n` definiert ist.

```
char *strtok(char *s, const char *t);
```

`s` wird nach Zeichenfolgen durchsucht, die durch Zeichen aus `t` begrenzt sind.

Die im folgenden beschriebenen Funktionen dienen zur Manipulation von Vektoren aus Zeichen (bei denen das `'\0'`-Zeichen nicht die Sonderbedeutung „Stringende“ besitzt!).

```
void *memcpy(void *s, const void *t, size_t n);
```

Kopiert die ersten `n` Zeichen von `t` nach `s`. Ergebnis ist `s`.

```
void *memmove(void *s, const void *t, size_t n);
```

Wie `memcpy`, funktioniert aber auch, wenn sich die Vektoren überlappen.

```
int memcmp(const void *s, const void *t, size_t n);
```

~~Vergleicht~~graphisch die ersten

unterbrochen werden und das Programm wird in der Funktion `fkt` an der Stelle des Aufrufs von `setjmp` fortgesetzt. Hierbei wird der in `env` gespeicherte Zustand wieder hergestellt und der Programmablauf in `fkt` geht so weiter, als ob `setjmp` gerade ausgeführt wurde und als Ergebnis den Wert des ganzzahligen Ausdrucks `expression` geliefert hat.

Üblicherweise steht der Aufruf von `setjmp` innerhalb der Bedingung einer `if`-Anweisung, `switch`-Konstruktion oder Schleifenkonstruktes.

11.9 Signale

Jede Implementierung legt in einer Voreinstellung fest, wie ein Programm auf Ausnahmesituationen (Interrupts, Signale — etwa Laufzeitfehler o.ä.) reagieren soll. Signale haben eine systemabhängige Nummer und die Signalbehandlung kann man sich so vorstellen, dass es eine Funktion vom Typ

```
void system_signal_behandlung(int);
```

(der Name ist frei erfunden) gibt, die beim Eintreffen eines Signals mit der Nummer des Signals als Argument aufgerufen wird. (Üblicherweise schreibt diese Funktion eine zur Signalnummer passende Meldung auf den Bildschirm und das Programm wird abgebrochen.)

Der Programmierer hat die Möglichkeit, eigene Signalbehandlungsfunktionen zu schreiben, die den gleichen Typ haben müssen, wie die systemeigene Signalbehandlungsfunktion.

Dem Programm muss dann nur noch mitgeteilt werden, dass es auf ein spezielles Signal nicht wie üblich, also mit dem Aufruf der systemeigenen Signalbehandlungsroutine, sondern, wie vom Programmierer gewünscht, mit dem Aufruf der eigenen Signalbehandlungsfunktion reagieren soll. Diese Mitteilung erfolgt mit der Funktion

```
void (*signal(int sig, void (*handler)(int)))(int);,
```

die in `signal.h` deklariert ist.

Das erste Argument ist die Nummer des Signals, für das die besondere Behandlung gewünscht wird; das zweite Argument ist ein Funktionszeiger auf die eigene Signalbehandlungsfunktion, die die besondere Behandlung durchführt.

Funktionsergebnis ist ein Zeiger auf die bisherige für das entsprechende Signal geltende Signalbehandlungsfunktion oder `SIG_ERR` im Fehlerfall.

Trifft nun das entsprechende Signal erstmalig ein, wird anstelle der systemeigenen Signalbehandlungsfunktion die eigene mit der Signalnummer als Argument aufgerufen. Anschließend wird der Programmablauf an der Stelle fortgeführt, an der das Signal auftrat. Bei nochmaligem Eintreffen des Signals wird wieder die systemeigene Behandlung des Signals durchgeführt.

Der Standard sieht explizit folgende Signale vor:

SIGABRT abnormaler Programmabbruch, etwa durch **abort**
SIGFPE Arithmetikfehler
SIGILL illegaler Maschinenbefehl
SIGINT Dialogsignal
SIGSEGV illegaler Speicherzugriff
SIGTERM Aufforderung von Außen, das Programm zu beenden

Ein Programm kann an sich selber mit

```
int raise(int sig);
```

Signale schicken. Im Fehlerfall ist das Funktionsergebnis von **raise** ungleich 0.

11.10 Datum und Uhrzeit

Funktionen zur Handhabung von Datum und Uhrzeit sind in **time.h** deklariert. Folgende, ebenfalls in **time.h** definierten Typen werden benötigt.

clock_t Arithmetischer Typ zur Darstellung von Zeitdifferenzen (etwa Rechenzeit). Darstellung und Einheit sind implementationsabhängig.
time_t Arithmetischer Typ zur Darstellung von Kalenderzeiten (Uhrzeit, Datum). Die Darstellung ist systemabhängig.
struct tm Struktur zur standardisierten Darstellung von Kalenderzeiten.

Die Komponenten der Struktur **struct tm** sind:

int tm_sec Sekunden nach der vollen Minute. Erlaubt sind Werte von 0 bis 61 (60 und 61 zur Realisierung von Schaltsekunden!).
int tm_min Minuten nach der vollen Stunde. Werte von 0 bis 59.
int tm_hour Stunden nach Mitternacht. Werte von 0 bis 23.
int tm_mday Tag im Monat. Werte von 1 bis 31.
int tm_mon Monate seit Januar. Werte von 0 bis 11.
int tm_year Jahre seit 1900.
int tm_wday Tage seit Sonntag. Werte von 0 bis 6.
int tm_yday Tage seit Neujahr. Werte von 0 bis 365.
int tm_isdst Kennzeichen für Sommerzeit. Positiv, falls Sommerzeit ist; 0, falls keine Sommerzeit ist und negativ, falls die entsprechende Information unbekannt ist.

Folgende Funktionen stehen zur Verfügung:

```
clock_t clock(void);
```

liefert die Rechenzeit seit Start des Programms (−1 im Fehlerfall).

Zur Umrechnung der systemspezifischen Einheit in **clock_t** in Sekunden steht das Makro **CLOCKS_PER_SEC** zur Verfügung. So liefert **clock()/CLOCKS_PER_SEC** die Rechenzeit in Sekunden.

```
time_t time(time_t *tp);
```

liefert die aktuelle Kalenderzeit (−1, falls wenn sie nicht bekannt ist!) und legt diese auch in ***tp** ab, falls **tp** ungleich **NULL** ist.

```
double difftime(time_t t2, time_t t1);
```

liefert die Differenz $t2 - t1$ in Sekunden.

```
time_t mktime(struct tm *tp);
```

wandelt die in der Struktur **tp* gespeicherte Kalenderzeit in den arithmetischen Typ *time_t* um, falls möglich. Sonst wird der Wert -1 geliefert.

```
char *asctime( const struct tm *tp);
```

wandelt die in **tp* gespeicherte Kalenderzeit um in eine Zeichenkette, die Datum und Uhrzeit in lesbarer Form enthält, legt diese Zeichenkette im statischen Speicher ab und gibt ihre Adresse als Wert zurück.

```
char *ctime( const time_t *tp);
```

macht dasselbe wie *asctime* mit der arithmetisch dargestellten Kalenderzeit **tp*.

```
struct tm *gmtime(const char time_t *tp);
```

wandelt die in **tp* arithmetisch abgespeicherte Kalenderzeit (Ortszeit) um in eine universelle Weltzeit (Greenwich-Mean-Time oder neuerdings Coordinated Universal Time). Funktionsergebnis ist *NULL*, falls dies nicht möglich ist (etwa, wenn die Information über Zeitzonen nicht zur Verfügung steht!).

```
struct tm *localtime(const char time_t *tp);
```

Gegenstück zu *mktime*; wandelt die in **tp* arithmetisch abgespeicherte Kalenderzeit in den Strukturtyp *struct tm* um.

Kalenderzeiten kann man nach eigenen Wünschen formatiert in eine Zeichenkette schreiben. Hierzu steht die Funktion

```
size_t strftime(char *s, size_t smax, const char *fmt, const struct tm *tp);
```

zur Verfügung. Unter Kontrolle des Formatstrings *fmt* wird die Kalenderzeit als Zeichenkette in das Feld *s* der Länge *smax* geschrieben. Die Anzahl der abgelegten Zeichen (ohne abschließendes Stringendezeichen $\backslash 0$) wird als Funktionsergebnis zurückgegeben bzw. 0, wenn der Platz nicht reicht.

Die Kontrolle durch den Formatstring funktioniert ähnlich wie bei den *printf*-Funktionen, d.h. Zeichen aus dem Formatstring, welche keine Formatanweisungen sind, werden einfach kopiert.

Es gibt folgende Formatanweisungen:

```
%c  Lokale Darstellung von Datum und Zeit.
%d  Tag im Monat (0–31).
%H  Stunde (00–23).
%I  Stunde (01–12).
%j  Tag im Jahr (001–366).
%m  Monat (01–12).
%M  Minute (00–59).
%p  lokales Äquivalent zu pm oder pm.
%S  Sekunde (00–61).
```

%U Woche im Jahr (Sonntag ist erster Wochentag) (00–53).
 %w Wochentag (0 für Sonntag, ..., 6 für Samstag).
 %W Woche im Jahr (Montag ist erster Wochentag) (00–53).
 %x Lokale Darstellung des Datums.
 %X Lokale Darstellung der Zeit.
 %y Jahr ohne Jahrhundertangabe (00–99).
 %Y Jahr mit Jahrhundert.
 %Z Zeitzone, falls bekannt.
 %% Ausgabe eines Prozentzeichens.

11.11 Systemabhängige Größen und Typen

In der Deklarationsdatei `stddef.h` sind die Typen `size_t` (Ergebnistyp des `sizeof`-Operators) und `ptrdiff_t` (ganzzahliger Typ, der groß genug ist, um die Differenz zweier Adresswerte aufzunehmen, siehe Abschnitt 13) definiert.

In der Datei `limits.h` sind symbolische Konstanten definiert, welche die ganzzahligen Wertebereiche festlegen. Die in der folgenden Tabelle angegebenen Werte sind Mindestwerte, können von einer Implementierung übertroffen werden:

CHAR_BIT	8	Bits in einen char
SCHAR_MAX	+127	maximaler Wert für signed char
SCHAR_MIN	−127	minimaler Wert für signed char
UCHAR_MAX	+255	maximaler Wert für unsigned char
CHAR_MAX		maximaler Wert für char also entweder UCHAR_MAX oder SCHAR_MAX
CHAR_MIN		minimaler Wert für char also entweder UCHAR_MIN oder 0
INT_MAX	+32767	maximaler Wert für int
INT_MIN	−32767	minimaler Wert für int
LONG_MAX	+2147483647	maximaler Wert für long int
LONG_MIN	−2147483647	minimaler Wert für long int
SHRT_MAX	+32767	maximaler Wert für short int
SHRT_MIN	−32767	minimaler Wert für short int
UINT_MAX	+65535	maximaler Wert für unsigned int
ULONG_MAX	+4294967295	maximaler Wert für unsigned long int
USHRT_MAX	+65535	maximaler Wert für unsigned short int

In `float.h` sind Konstanten zur Gleitpunktdarstellung und Gleitpunktarithmetik definiert. Mindestens die in folgender Tabelle angegebenen Kenngrößen existieren. In der Tabelle angegebenen Werte sind Mindestgrößen:

FLT_RADIX	2	Basis der Exponentendarstellung
FLT_ROUNDS		Art der Rundung bei Gleitpunktaddition
FLT_DIG	6	dezimale Genauigkeit für <code>float</code>
FLT_EPSILON	$1E-5$	Maschinengenauigkeit, kleinste <code>float</code> -Zahl x mit $1.0f + x \neq 1.0f$
FLT_MANT_DIG		Anzahl der Mantissenstellen
FLT_MAX	$1E+37$	maximaler <code>float</code> -Wert
FLT_MAX_EXP		größter ganzzahliger Exponent n , so dass $\text{FLT_RADIX}^n - 1$ als <code>float</code> darstellbar ist
FLT_MIN	$1E-37$	kleinster normalisierter <code>float</code> -Wert
FLT_MIN_EXP		kleinster ganzzahliger Exponent n , so dass FLT_RADIX^n normalisiert als <code>float</code> dargestellt werden kann
DBL_DIG	10	dezimale Genauigkeit für <code>double</code>
DBL_EPSILON	$1E-9$	Maschinengenauigkeit, kleinste <code>double</code> -Zahl x mit $1.0 + x \neq 1.0$
DBL_MANT_DIG		Anzahl der Mantissenstellen
DBL_MAX	$1E+37$	maximaler <code>double</code> -Wert
DBL_MAX_EXP		größter ganzzahliger Exponent n , so dass $\text{FLT_RADIX}^n - 1$ als <code>double</code> darstellbar ist
DBL_MIN	$1E-37$	kleinster normalisierter <code>double</code> -Wert
DBL_MIN_EXP		kleinster ganzzahliger Exponent n , so dass FLT_RADIX^n normalisiert als <code>double</code> dargestellt werden kann

12 Strukturen

12.1 Einleitung

In Feldern sind mehrere einzelne Variablen zu einer Einheit zusammengefasst und alle diese Variablen (Datenkomponenten) haben den gleichen Typ — etwa in

```
int a[100];      und
double m[5][10];
```

ist `a` ein eindimensionales Feld der Länge 100 vom Typ `int`, d.h. es sind 100 `int`-Variablen (alle haben den Typ `int`) definiert mit den Namen `a[0]`, ..., `a[99]` und `m` ist ein zweidimensionales Feld, in dem 50 `double`-Werte (alle haben den Typ `double`) abgelegt werden können. Diese `double`-Variablen haben die Namen `m[0][0]`, ..., `m[4][9]`.

In der heutigen Datenverarbeitung ist es üblich, auch mehrere Daten von unterschiedlichem Typ zu einer Einheit zusammenzufassen, wenn diese Daten in einem inneren Sinnzusammenhang zueinander stehen.

Will man etwa in einem Personalverwaltungsprogramm die Adressen von Mitarbeitern verarbeiten, so hat eine (Mitarbeiter-)Adresse mehrere Bestandteile:

- die Straße (könnte in einem `char`-Feld als String abgelegt werden),
- eine Hausnummer (könnte als `int`-Wert abgespeichert werden),
- eine Postleitzahl (könnte wiederum als `int`-Wert abgespeichert werden) und schließlich
- einen (wieder in einem `char`-Feld als String abzuspeichernden) Wohnort.

Aus Sicht der Anwendung (Personalverwaltung) bilden diese vier Bestandteile als ganzes eine *Einheit* (nämlich die *Adresse* eines Mitarbeiters).

In der Programmiersprache C heißt die Realisierung einer solchen, aus mehreren, möglicherweise vom Typ her unterschiedlicher Bestandteile bestehende Einheit *Strukturtyp* und die Definition eines solchen Strukturtypes geschieht wie folgt:

```
struct adresse {
    char strasse[64];
    int hausnr;
    int plz;
    char ort[64];
};
```

Hierdurch ist zunächst ein “neuer” Typ vereinbart und noch kein einziges Byte Speicher zur Abspeicherung für irgendwas reserviert.

Es ist nur erreicht, dass es einen neuen Typen mit dem Namen `struct adresse` gibt und dass dieser Typ aus vier Bestandteilen (*Komponenten* genannt) besteht:

- die erste Komponente mit dem Namen `strasse` — diese Komponente ist ein `char`-Feld der Länge 64,
- die zweite Komponente mit dem Namen `hausnr` — diese Komponente hat den Typen `int`,
- die dritte Komponente mit dem Namen `plz` — auch diese Komponente hat den Typen `int` — und

- die vierte Komponente mit dem Namen `ort` — diese Komponente ist ebenfalls ein `char`-Feld der Länge 64.

Von diesem Typ können nun wie üblich Variablen definiert werden:

```
struct adresse myaddr, chefaddr;
```

Das Objekt `myaddr` ist eine Variable vom Typ `struct adresse` und ist groß genug, um alle bei der Typvereinbarung angegebenen Komponenten aufnehmen zu können (auf unseren Rechnern belegt die Variable `myaddr` (höchstwahrscheinlich) 136 aufeinanderfolgende Bytes, 64 für die erste Komponente `strasse` (Feld der Länge 64 vom Typ `char`), jeweils 4 Byte für die zweite und dritte Komponente `hausnr` und `plz` (jeweils vom Typ `int`) und schließlich nochmal 64 Byte für die Komponente `ort` (Feld der Länge 64 vom Typ `char`).

Das Objekt `chefaddr` ist ein anderes Objekt vom gleichen Typ, belegt also wiederum (etwa) 136 Byte anderen Speicherbereich.

Wird der Strukturtyp nur einmal zur Definition von Variablen benötigt, so kann dies sofort bei der Definition des Types erfolgen, wobei dann hinter dem Schlüsselwort `struct` kein weiterer Name angegeben zu werden braucht (aber kann):

```
struct {
    char strasse[64];
    int hausnr;
    int plz;
    char ort[64];
} myaddr, chefaddr;
```

(Wird wie hier kein Name angegeben, so kann auf den Typen später jedoch nicht mehr zugegriffen werden, da er keinen Namen hat!)

Hat man eine Variable vom Strukturtyp definiert (die Variable “ist“ dann eine Struktur), so kann man mit dem Punktoperator `.` auf die Komponenten der Struktur zugreifen.

Ist etwa (wie oben definiert) die Variable `myaddr` vom Typ `struct adresse`, so hat diese Variable 4 Komponenten:

- `myaddr.strasse` vom Typ `char []`, also ein Zeichenfeld,
- `myaddr.hausnr` vom Typ `int`,
- `myaddr.plz` ebenfalls vom Typ `int` und
- `myaddr.ort` wiederum vom Typ `char []`.

Die “Objekte“ `myaddr.hausnr` und `myaddr.plz` kann man wie jede andere `int`-Variable verwenden und `myaddr.strasse` sowie `myaddr.ort` wie jedes andere Zeichenfeld, etwa:

```
myaddr.hausnr = 51; /* die Komponente hausnr der
                    Struktur myaddr wird 51 */
printf("Strasse: %s\n", myaddr.strasse); /* die Komponente strasse
                    der Struktur myaddr wird ausgegeben */
...
```

Natürlich kann man auch Felder von Strukturen definieren:

```
struct adresse mitarbeiter_adressen[10];
```

Das Objekt `mitarbeiter_adressen` ist ein Feld der Länge 10, wobei jedes Feldelement eine Struktur vom Typ `struct adresse` ist. (Dieses Feld belegt dann $10 \cdot 136 = 1360$ Byte!) Das erste Feldelement ist `mitarbeiter_adressen[0]` und das letzte ist `mitarbeiter_adressen[9]`. Ist der Index `i` größer gleich 0 und kleiner als 10, so kann man auf die Komponenten des entsprechenden Feldelementes wie folgt zugreifen:

```
mitarbeiter_adressen[i].strasse    bzw.
mitarbeiter_adressen[i].hausnr    bzw.
mitarbeiter_adressen[i].plz       bzw.
mitarbeiter_adressen[i].ort
```

Eine Variable vom Strukturtyp kann bei ihrer Definition explizit vorbesetzt werden:

```
struct adresse myaddr = { "Karl-Arnold-Ring", 51, 52457, "Aldenhoven" };
```

(D.h. in geschweiften Klammern muss dann eine Liste von initialisierenden Ausdrücken von den passenden Typen stehen! Eine nicht explizit initialisierte Variable vom Strukturtyp wird, wie andere Variablen auch, nicht implizit initialisiert, wenn es sich um eine automatische Variable handelt bzw. mit lauter Nullbytes initialisiert, wenn es sich um eine externe oder statische Variable handelt!)

Im Gegensatz zu Feldern ist für Strukturen der (einfache) Zuweisungsoperator `=` erlaubt, es erfolgt eine bitweise Kopie der einzelnen Komponenten. Dies gilt auch für Komponenten, welche selber Felder sind! Eine Zuweisung:

```
chefaddr = myaddr;
```

ist somit erlaubt — hier werden alle Komponenten von `myaddr` in die entsprechenden Komponenten von `chefaddr` kopiert, insbesondere werden auch Strassennamen und Wohnort übernommen (beides sind `char`-Felder!).

Die Vergleichsoperatoren `==` und `!=` hingegen sind auch für Strukturen nicht erlaubt!

Es ist nicht zwingend vorgeschreiben, dass die Komponenten eines Strukturtypes unterschiedliche Typen besitzen müssen — zur Bearbeitung von Bildschirmpunkten könnte auch folgende Struktur sinnvoll verwendet werden:

```
struct bildschirmpunkt {
    int ix; /* Abstand vom oberen Bildschirmrand */
    int iy; /* Abstand vom unteren Bildschirmrand */
};
```

Der Typname eines Strukturtypen besteht (wie bei Aufzählungstypen) aus zwei Worten — mittels `typedef` kann man dann einen neuen, nur aus einem Wort bestehenden Typnamen vereinbaren, etwa:

```
typedef struct adresse Adresse;
```

den man dann wie folgt etwa zur Variablendefinition verwenden kann:

```
Adresse myaddr, chefaddr;
```

Wird ein Strukturtyp in mehreren Quelltexten eines umfangreicheren Programms benötigt, so muss dieser Typ natürlich in jedem der Quelltexte bekannt sein — d.h. in jedem Quelltext aufs neue definiert werden. Hat man dem Strukturtyp mittels **typedef** einen neuen Namen gegeben und will man diesen Namen in mehreren Quelltexten verwenden, so muss man nicht nur den Strukturtyp, sondern auch den neuen Namen in jedem Quelltext erneut definieren.

Auch hier bietet sich an, die Typdefinition (ohne Variablenvereinbarung) und ggf. das **typedef** in eine eigene Header-Datei zu schreiben und diese überall wo notwendig einzubinden.

12.2 Adress-Variablen vom Strukturtyp

Ein Strukturtyp kann wie jeder andere Typ verwendet werden, man kann etwa einfache Variablen von diesem Typ definieren:

```
struct adresse myaddr;
```

oder, wie bereits gesehen, Felder vom Strukturtyp:

```
struct adresse mitarbeiter_adressen[10];
```

oder aber auch Adress-Variablen vom Strukturtyp:

```
struct adresse *addrptr;
```

Zu beachten ist, dass durch diese Vereinbarung der Adress-Variablen **addrptr** vom Typ **struct adresse** nicht der Speicherplatz für eine ganze **struct adresse** — also (etwa) 136 Byte — reserviert wird, sondern nur der Speicherplatz für eine (Speicher-)Adress-Variable (Zeiger, Pointer) — also 4 oder 8 Byte. Mit dieser Adress-Variablen **addrptr** kann erst dann sinnvoll gearbeitet werden, wenn dieser ein entsprechender Adress-Wert zugewiesen worden ist, etwa:

```
addrptr = &myaddr;
```

Ist dies geschehen, so kann dann wie üblich mit dem Verweis-Operator auf das **struct adresse**-Objekt zugegriffen werden, dessen Adresse in **addrptr** abgelegt ist. Nach obiger Zuweisung ist etwa

```
(*addrptr).strasse
```

der Straßenname der in **myaddr** abgespeicherten **struct adresse** und

```
(*addrptr).plz
```

die dort stehende Postleitzahl. Man beachte, dass aufgrund der C-Auswertungsregeln (vgl. Opeartortabelle auf Seite 46 bzw. 138) die runden Klammern um ***addrptr** stehen müssen, da der Verweis-Operator ***** eine niedrigere Priorität als der "Komponentenzugriffsoperator" **.** hat.

Ohne Klammerung würde zuerst der Punktoperator **.** und dann erst der Verweis-Operator ***** "greifen". Dies würde hier zu einem Syntaxfehler führen, denn die Variable **addrptr** ist selbst keine Strukturvariable (sondern eine Adress-Variable vom Strukturtyp) und hat somit selbst keine Komponenten (Komponenten hat allenfalls das Objekt, dessen Speicher-Adresse in **addrptr** abgelegt ist) und demnach kann auf

`addrptr` der Punktoperator `.` nicht angewendet werden.

Zur Vermeidung solcher Klammerungen gibt es in C einen weiteren “Komponentenzugriffsoperator” `->`, dessen linker Operand aber eine Adress-Variable vom Strukturtyp sein muss und dessen rechter Operand ein Komponentename der entsprechenden Struktur ist.

Der Zugriff:

```
addrptr->hausnr
```

ist vollkommen gleichwertig zu

```
(*ptraddr).hausnr
```

und beide sind nur dann sinnvoll, wenn in der (Speicher-)Adress-Variablen `addrptr` eine gültige Adresse eines `struct adresse`-Objektes abgelegt ist.

12.3 Funktionsargumente und –ergebnisse vom Strukturtyp

12.3.1 Strukturen als Funktionsargumente

Man kann Strukturen an Funktionen als Argumente übergeben. Die Funktion muss entsprechend definiert sein, d.h. sie muss eine entsprechende Parametervariable haben.

Eine wie folgt deklarierte:

```
void fkt( struct adresse );
```

und definierte:

```
void fkt( struct adresse a) { ... }
```

Funktion kann dann entsprechend aufgerufen werden:

```
... fkt( myaddr )...;
```

wobei hier wiederum das reine Prinzip *Call by Value* realisiert ist, d.h. die als Argument übergebene Struktur `myaddr` wird in die lokale Strukturvariable `a` kopiert (alle Komponenten — insgesamt also etwa 136 Byte — werden bitweise übertragen). Beim Ablauf der Funktion gibt es somit zwei Strukturen mit (zunächst) gleichen Komponenten — im Verlauf der Funktion kann aber die lokale Kopie (`a`) abgeändert werden, ohne dass das Original (`myaddr`) der aufrufenden Funktion dadurch verändert wird. Hat eine Struktur viele große Komponenten, so ist für das System bei einem solchen Aufruf einige Kopierarbeit notwendig.

Aus diesem Grund ist es manchmal günstiger (Speicherplatzersparnis und Zeitersparnis), dass man Funktionsargumente vom Strukturtyp nicht direkt übergibt, sondern deren Adressen. Die Funktion muss dann natürlich entsprechend deklariert:

```
void fkt( struct adresse *);
```

und definiert:

```
void fkt( struct adresse *ap) { ... }
```

werden. Der Aufruf muss dann wie folgt lauten:

```
... fkt( &myaddr )...;
```

Die lokale Parametervariable der Funktion ist jetzt eine (Speicher-)Adress-Variable vom Strukturtyp, hat als Speicher-Adress-Variable nur etwa 4 (oder 8) Byte Speicher nötig und in diese Speicher-Adress-Variable wird bei obigem Aufruf die Adresse der `struct adresse`-Variablen `myaddr` hineingeschrieben. Es gibt also nur das eine `struct adresse`-Objekt `myaddr`, das Umkopieren entfällt. (Natürlich hat die Funktion jetzt Zugriff auf die originale Struktur des aufrufenden Programnteils und sollte diese nicht leichtfertig verändern — derartige Veränderungen werden an der originalen Variablen durchgeführt! Um derartige, leichtfertige Veränderungen der mittels Adresse als Funktionsargument übergebenen Struktur zu verhindern, wäre in diesem Fall folgende Deklaration

```
void fkt( const struct adresse *ap) { ... }
```

und Definition

```
void fkt( const struct adresse *ap) { ... }
```

der Funktion besser, vgl. Abschnitt 7.4.2!)

12.3.2 Strukturen als Funktionsergebnisse

Strukturen sind auch als Funktionsergebnisse möglich. D.h. man könnte etwa eine Funktion wie folgt deklarieren:

```
struct adresse fkt(...);
```

wobei bei deren Definition eine entsprechende Struktur hinter dem `return` anzugeben ist:

```
struct adresse fkt(...)
{
    struct adresse tmp;
    .
    .
    .
    return tmp;
}
```

Diese Funktion könnte etwa wie folgt aufgerufen werden:

```
myaddr = fkt (...);
```

wobei der Struktur-Variablen `myaddr` ein neuer Wert zugewiesen wird (Zuweisung von Strukturen ist möglich!), nämlich der durch die aufgerufene Funktion zurückgegebene.

Doch auch hier ein Hinweis auf das, was intern abläuft (vgl. die Diskussion von Funktionsergebnissen auf Seite 103):

Kurz vor dem Ende dieser Funktion gibt es zunächst zwei Objekte vom Typ `struct addr`, nämlich die zur Funktion `fkt` lokale Variable `tmp`, deren Wert ja zurückgegeben werden soll, und die Variable `myaddr` des aufrufenden Programnteils, welcher das Funktionsergebnis zugewiesen werden soll!

Jedoch zu dem Zeitpunkt, wo im aufrufenden Programmteil die Zuweisung des Funktionsergebnisses stattfinden soll, ist die Funktion ja gerade beendet und die lokale Variable `tmp` existiert dann ja bereits nicht mehr! Damit die Zuweisung jedoch funktioniert, wird das Funktionsergebnis vom System in einem temporären Zwischenspeicher abgelegt, die Zuweisung durchgeführt (bzw. das Funktionsergebnis weiter verarbeitet, wenn der Funktionsaufruf in einem komplexeren Ausdruck statgefunden hatte) und nach der Zuweisung wird dieser temporäre Zwischenspeicher wieder freigegeben. D.h. es gibt dann kurzzeitig sogar drei beteiligte Strukturen, neben den beiden oben erwähnten `tmp` und `myaddr` noch die “systemeigene“ temporäre, in der das Funktionsergebnis kurzzeitig abgelegt wird.

Dies ist auch bei Funktionsergebnissen irgendeines anderen Typen der Fall, das Ergebnis wird in einem “systemeigenen“ temporären Bereich zwischengespeichert!

Bei großen Strukturen ist jedoch der benötigte Speicherplatz zu bedenken. In unserem Beispiel sind dann am Ende der Funktion 3 mal 136 Byte belegt! (Jeweils 136 Byte für die Variable `tmp` der Funktion, 136 Byte für das temporäre Funktionsergebnis und 136 Byte für die Variable `myaddr` des aufrufenden Programmteils!)

Insbesondere bei noch größeren Strukturtypen kostet dies Rechenzeit und Speicherplatz.

Dies kann vermieden werden, indem man wiederum keine Werte vom Strukturtyp zurückgibt, sondern Adressen von Struktur-Objekten. Man muss dann die Funktion entsprechend deklarieren, definieren und aufrufen und dafür sorgen, dass das Objekt, dessen Adresse zurückgegeben wird, nach dem Funktionsende noch vorhanden ist (also keine Rückgabe von Adressen lokaler Objekte!). Ein Beispiel könnte wie folgt aussehen, Deklaration:

```
struct adresse * fkt ( struct adresse * , ... );
```

Definition:

```
struct adresse * fkt ( struct adresse *a , ... )
{ .
  .
  .
  return a;
}
```

und Aufruf:

```
myaddr = * fkt( &chefaddr, ... );
```

Hier existieren nur die beiden Struktur-Objekte `myaddr` und `chefaddr` des Hauptprogramms (und ein Paar Speicher-Adress-Variablen mit jeweils 4 Byte Größe) und es werden keine zusätzlichen lokalen oder temporären Struktur-Variablen erzeugt.

Durch die Rückgabe des Parameters ist hier sichergestellt, dass das Struktur-Objekt, dessen Adresse durch die Funktion zurückgegeben wird, nach Funktionsende noch vorhanden ist. Auf das Funktionsergebnis wird hier der Verweis-Operator `*` angewendet, wodurch auf das Objekt `chefaddr` verwiesen wird.

12.4 Struktur-Komponenten vom Strukturtyp

Komponenten einer Struktur können wiederum selbst von einem (anderen) Strukturtyp sein, etwa:

```
struct angestellter {
    char v_name[32];
    char n_name[64];
    struct adresse addr;
    char genus;
    long pers_nr;
    ...
};
```

Die dritte Komponente `addr` dieses neuen Strukturtypes `struct angestellter` ist vom (vorher bereits bekannt sein müssenden) Strukturtyp `struct adresse`.

Definiert man eine Variable von diesem neuen Strukturtyp:

```
struct angestellter chef;
```

so ist die erste Komponente `chef.v_name` der Vorname (Zeichenfeld der Länge 32), die zweite Komponente `chef.n_name` der Nachname (Zeichenfeld der Länge 64), die dritte Komponente `chef.addr` die Adresse (Struktur vom Typ `struct adresse` — diese Komponente belegt selbst wiederum etwa 136 Byte Speicher und in diesen 136 Byte sind Daten unterschiedlichen Types wie Strasse, Hausnummer usw. abgelegt) usw. des entsprechenden in `chef` abgespeicherten Angestellten.

Auf die (Unter-)Komponenten der dritten Komponente wird wie folgt zugegriffen:

- `chef.addr` ist die komplette Adresse.
- `chef.addr.strasse` ist der in der Adresse abgelegte Strassenname (also ein Zeichenfeld der Länge 64),
- `chef.addr.hausnr` ist die (int) Hausnummer,
- `chef.addr.plz` ist die (int) Postleitzahl und
- `chef.addr.ort` ist der Wohnort (auch ein Zeichenfeld der Länge 64).

Ist `ang_ptr` eine Adress-Variable vom Typ `struct angestellter` — diese müsste wie folgt definiert worden sein:

```
struct angestellter *ang_ptr;
```

und ist in dieser (Speicher-)Adress-Variablen eine gültige Speicheradresse (etwa nach der Zuweisung: `ang_ptr = &chef;`) abgelegt, so kann mit dem Punktoperator `.` (Klammern nicht vergessen) oder mit dem Operator `->` entsprechend zugegriffen werden:

- `(*ang_ptr).addr` oder `ang_ptr->addr`
ist die komplette Adresse,
- `(*ang_ptr).addr.strasse` oder `ang_ptr->addr.strasse`
ist der Strassenname,
- `(*ang_ptr).addr.hausnr` oder `ang_ptr->addr.hausnr`
ist die Hausnummer,

- `(*ang_ptr).addr.plz` oder `ang_ptr->addr.plz`
ist die Postleitzahl und
- `(*ang_ptr).addr.ort` oder `ang_ptr->addr.ort`
ist der Wohnort.

Bei der Definition von Strukturtypen, die selbst wieder Komponenten vom Strukturtyp besitzen, ist darauf zu achten, dass keine indirekte oder direkte Rekursion entsteht, d.h.

- hat ein Strukturtyp `struct A` eine Komponente vom Strukturtyp `struct B`, so darf der Strukturtyp `struct B` selbst keine Komponente vom Strukturtyp `struct A` haben (indirekte Rekursion),
- ein Strukturtyp `struct A` darf keine Komponente vom selben Strukturtyp `struct A` haben (direkte Rekursion).

(In beiden Fällen würde eine entsprechende Struktur-Variable unendlich viel Speicher benötigen!)

12.5 Speicherlücken, Alignment

Wendet man den `sizeof`-Operator auf Strukturtypen oder Variablen vom Strukturtyp an, so kann es bisweilen sein, dass der Ergebniswert anders als erwartet ist!

Dies liegt dann an der maschinenabhängigen Organisation des Arbeitsspeichers:

Auf den meisten Rechnern, auf denen etwa ein `int`-Wert mit 4 Byte abgespeichert wird, muss die Adresse einer `int`-Variable immer durch 4 teilbar sein (eine derartige Teilbarkeitsvorschrift für Speicheradressen heißt *Alignment*).

Definiert man dann etwa einen Strukturtyp und ein entsprechendes Feld wie folgt:

```
struct A {
    int i;
    char c;
} A_feld[2];
```

so muss die Adresse von `A_feld[0]` durch 4 teilbar sein, da `A_feld[0]` mit der `int`-Komponente `i` beginnt, und die Adresse von `A_feld[1]` ebenfalls — ist etwa 1000 die Speicheradresse von `A_feld[0]` (1000 ist durch 4 teilbar!), so werden die Bytes mit den Nummern 1000 bis 1003 für die `i`-Komponente von `A_feld[0]` verwendet und das folgende Byte mit der Nummer 1004 für die `c`-Komponente. Die nächste durch 4 teilbare freie Speicheradresse ist dann 1008 und an dieser Stelle kann dann frühestens `A_feld[1]` (mit seiner `int`-Komponente) beginnen — d.h. die Bytes mit den Nummern 1005 bis 1007 werden für `A_feld` nicht verwendet und können auch anderweitig (etwa für andere Variablen) nicht verwendet werden ("Speicherlücke")! Obwohl ein Objekt vom Typ `struct A` nur 5 Byte benötigt, müssen etwa in Feldern hierfür intern 8 Byte verwendet werden.

Der `sizeof`-Operator liefert dann für ein `struct A`-Objekt (und zwar unabhängig davon, ob es sich um ein Element eines Feldes handelt oder auch nicht) diese 8 (Byte) als Ergebnis und für ein entsprechendes Feld der Länge 2 das Ergebnis 16 — d.h.

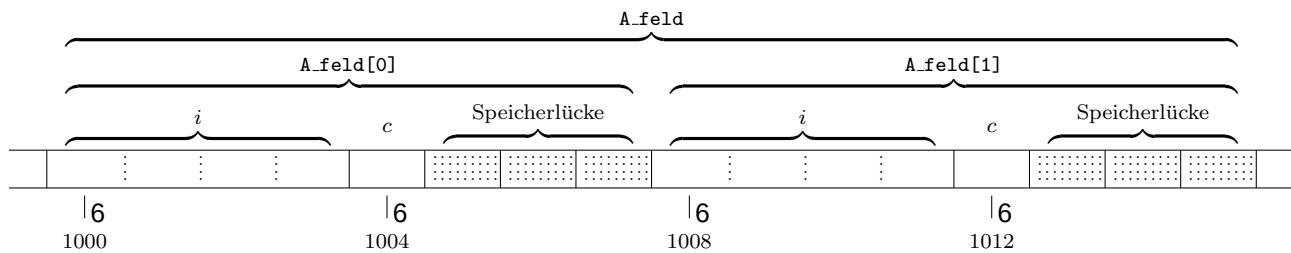


Abbildung 26: Speicherlücken

die maschinenabhängige Implementierung des `sizeof`-Operators berücksichtigt die maschinenabhängigen *Alignment-Vorschriften*.

12.6 Selbstreferierende Strukturen

Obwohl ein Strukturtyp keine Komponente vom eigenen (Struktur-)Typ haben darf, kann ein Strukturtyp eine (oder auch mehrere) Adress-Variable(n) vom eigenen Typen haben, etwa:

```
struct A {
    ...
    struct A * next;
};
```

Die Komponente `next` ist eine Adress-Variable vom "eigenen" Typ `struct A`, d.h. für diese Adress-Variable sind (unabhängig von der Größe des zugrundeliegenden Types) `struct A` etwa (4 oder 8) Byte Speicher notwendig und in dieser Adress-Variablen kann die Adresse eines Objektes vom selben Typ (oder der Adress-Wert `NULL`) abgelegt werden. D.h. ein Objekt vom Typ `struct A` hat eine Komponente, die selbst wiederum auf ein Objekt vom gleichen Typ `struct A` "zeigen" kann.

Eine derartige Struktur heißt auch *rekursive Struktur*.

Mittels solcher rekursiver Strukturen lassen sich die heutzutage in der Datenverarbeitung häufig verwendeten, sogenannten *dynamischen Datenstrukturen* wie *Lineare Listen*, *doppelt verkettete Listen*, *Binärbäume* etc. realisieren. Typische Problemstellungen im Zusammenhang mit solchen Strukturen lassen sich dann häufig sehr elegant mittels rekursiver Funktionen und dynamischer Speicherverwaltung (Bibliotheksfunktionen `malloc` und `free`) erledigen.

Ein Beispiel zum Aufbau, Ausgabe und Löschen einer linearen Liste ist im Programm `struktur/liste/liste.c` nachzulesen.

12.7 Unions

Eine Union in C ist eine Variable, in der wahlweise (nicht gleichzeitig) Werte von unterschiedlichem Typ abgelegt werden können.

Durch

```
union wert
{ int ival;
  float fval;
  char *sval;
};
```

```
union wert u;
```

wird eine Variable `u` definiert, die wahlweise einen `int`-Wert, einen `float`-Wert oder eine Adresse einer Zeichenvariablen aufnehmen kann. (Es wird nur der Speicherplatz bereitgestellt, der ausreicht, um das größte der beteiligten Objekte aufzunehmen!)

Durch

```
u.ival = 32;
```

wird der `int`-Wert 32 in `u` gespeichert und durch

```
u.fval = 3.14;
```

der `float`-Wert 3.14.

Syntakisch werden Unions behandelt wie Strukturen.

Der Programmierer muss selbst verfolgen, welcher Typ als letztes in der Union abgelegt wurde. (Insbesondere ist nicht definiert, was z.B. passiert, wenn man auf die Variable `u`, die als letztes einen `int`-Wert aufgenommen hatte, mittels `u.float` zugreift.)

Wird eine Variable vom Uniontyp bei ihrer Erzeugung explizit initialisiert, so kann sie nur in dem Typen initialisiert werden, zu dem ihre "erste" Komponente gehört. In unserem Beispiel kann eine Variable vom Typ `union wert` nur mit einem `int`-Wert initialisiert werden!

12.8 Bitfelder

Ein Bitfeld ist eine Menge von nebeneinander stehenden Bits innerhalb einer implementationsabhängigen Speichereinheit (Maschinenwort).

Syntax für die Definition und den Zugriff ist die gleiche wie für Strukturen, etwa:

```
struct status
{ unsigned int genus : 1 ;
  unsigend int familienstand : 2 ;
  unsigned int anzahl_kinder : 3 ;
  unsigned int steuerklasse : 3 ;
};
```

```
struct status merkmale;
```

Die Variable `merkmale` enthält 4 Bitfelder mit den Namen `genus`, `familienstand`, `anzahl_kinder` und `steuerklasse`.

Die Zahl hinter dem Doppelpunkt gibt jeweils die Feldbreite (Anzahl der Bits) des einzelnen Bitfeldes an. Um sicherzustellen, dass es sich um vorzeichenlose Größen handelt, müssen Bitfelder als `unsigned int` vereinbart werden.

In `merkmale.genus` können die Werte 0 oder 1 abgelegt werden (etwa 0 für männlich und 1 für weiblich).

In `merkmale.familienstand` können die Werte 0,1,2 oder 3 abgelegt werden (zur Verschlüsselung des Familienstandes sollten 4 Werte reichen!).

In `merkmale.anzahl_kinder` und `merkmale.steuernummer` können jeweils die Werte 0 bis 7 eingetragen werden.

Es gibt keine Vektoren von Bitfeldern und der Adress-Operator kann nicht auf Bit-Felder angewendet werden.

Fast alles zu Bitfeldern ist implementationsabhängig.

13 Mehr zu Adressen

13.1 Adress-Arithmetik

Neben den Verweis-Operatoren `*` bzw. `->` bei Adressen von Strukturen und dem Cast-Operator zur Umwandlung von Adresstypen sind einige arithmetische Operatoren unter gewissen Einschränkungen für Adress-Werte zugelassen. (Auf Adress-Variablen ist natürlich auch der Adress-Operator `&` anwendbar!)

Diese Operatoren und ihre Effekte auf Operanden vom Adress-Typ werden im Folgenden aufgelistet. Hierzu seien `p` und `q` zwei Adress-Werte bzw. Zeiger vom gleichen Typ, d.h. sie können Adressen von Variablen gleichen Types aufnehmen.

- (a) Die relationalen Operatoren `<`, `>`, `<=`, `>=`, `==` und `!=` sind für `p` und `q` erlaubt, falls `p` und `q` auf die Elemente desselben Feldes "zeigen".

Beispiel:

```
int i, j, s[100], *p, *q;
.
.
.
p = &s[i];
q = &s[j];
if ( p < q )
.
.
.
```

`p < q` ist hierbei genau dann *wahr*, falls `p` auf ein Element des Feldes `s` zeigt, welches einen kleineren Index besitzt als das Element, auf das `q` zeigt (d.h. `*p` „steht links von“ `*q`).

Hierbei ist auch erlaubt, dass `p` oder `q` den Wert der ersten Adresse hinter dem Feldende besitzt, wenn also im obigen Beispiel `p` oder `q` auf „`s[100]`“ zeigen würde (beachte: das Feld `s` der Länge 100 beginnt mit `s[0]` und endet mit `s[99]!`).

Der Vergleich (mit `==` und `!=`) ist immer erlaubt, wenn `p` oder `q` den symbolischen Wert `NULL` hat.

- (b) Die logische Negation `!` ist für Adress-Werte zugelassen. Aus einem Adress-Wert ungleich `NULL` wird hierbei der Wert `NULL` und aus dem Wert `NULL` wird der Wert `1`.
- (c) Addition und Subtraktion ganzzahliger Werte auf bzw. von Adress-Werte(n) sind zulässig, falls es sich nicht um typlose Adressen handelt!

Der Effekt derartiger Operationen bedarf einer Erläuterung:

Zeigt der Adress-Wert `p` auf ein Element eines Feldes, so zeigt der Zeigerwert `p+1` auf das nächst folgende Element des Feldes, `p+2` auf das übernächste und `p-1` auf das vorherige etc..

Dies ist unabhängig vom Typ der Feldelemente, d.h. die Änderung des Adresswertes berücksichtigt den Typ der Objekte, auf die der Zeiger verweist.

Hypothetisches Beispiel:

`int`-Größen belegen auf unserer Beipielmaschine 4 Byte und es sei `ip` ein Zeiger auf `int` sowie `i` ein `int`-Feld der Länge 100 und `i[10]` habe die Adresse 1000. Nach der Zuweisung

```
ip = &i[10];
```

hat `ip` den Wert 1000 (dies ist kein Wert vom Typ `int` oder `long` o.ä., sondern vom Typ `int *`, Zeiger auf `int`!).

`ip+1` hat dann den Wert 1004, `ip+1` zeigt also auf die `int`-Variable, welche hinter `i[10]` im Kernspeicher steht.

Belegen nun `double`-Größen 8 Byte auf unserer Maschine und sei weiter `dp` ein Zeiger auf `double` sowie `d` ein `double`-Feld der Länge 100 und `d[10]` habe die Adresse 2000, so hat nach der Zuweisung

```
dp = &d[10];
```

`dp` den Wert 2000 (vom Typ `double *`, Zeiger auf `double`).

`dp+1` hat dann den Wert 2008, zeigt also auf die `double`-Variable, die hinter `d[10]` im Speicher steht.

- (d) Falls es sich wiederum nicht um typlose Adressen handelt, ist die Subtraktion von Adress-Werten, die auf Elemente eines (desselben!) Feldes (oder auf das erste Element „hinter“ dem Feld) zeigen, erlaubt. Das Ergebnis von `p - q` ist die Anzahl (mit negativem Vorzeichen, falls `p` kleiner als `q` ist) der Feldelemente, die zwischen `*p` (exklusive) und `*q` (inklusive) stehen.

Auch hier wird wieder implizit die Größe der Feldelemente berücksichtigt.

In Abschnitt 6.7 haben wir bereits kennengelernt, dass Adress-Werte und Adress-Variablen wie Feldnamen verwendet werden können (funktioniert zur Laufzeit aber nur, wenn die Adresse auf ein entsprechendes Feld „zeigt“!): Ist etwa `p` eine `int`-Adress-Variable und zeigt `p` auf den Anfang eines Feldes, so ist `p[0]` das erste Feldelement, `p[1]` das zweite usw.. Derartige Zugriffe auf Feldelemente kann man äquivalent auch wie folgt erreichen:

`*p` entspricht `p[0]`, also dem ersten Feldelement

`*(p+1)` entspricht `p[1]`, also dem zweiten Feldelement

`*(p+2)` entspricht `p[2]`, also dem dritten Feldelement usw..

Zeigt `p` nicht auf den Anfang, sondern auf die „Mitte“ eines Feldes, so sind auch folgende (syntaktisch korrekten) Zugriffe zur Laufzeit möglich:

`*(p-1)` entspricht `p[-1]`, also dem Feldelement, welches vor dem steht, auf das

`p` im Augenblick zeigt.

`*(p-2)` entspricht `p[-2]`, also dem Feldelement vor `p[-1]` usw..

Die Klammern etwa in `*(p+1)` und `*(p-1)` sind aufgrund der Operator-Vor-rangregeln (vgl. Tabellen auf Seite 46 und 138) notwendig, in `*p+1` würde die 1 auf `*p` addiert werden, also nicht auf die Adresse, sondern auf das, worauf die Adresse verweist!

- (e) Zuweisung von Zeigern gleichen Types ist natürlich erlaubt.

Bei Typungleichheit muss (mit Ausnahme bei `void *`-Adress-Variablen auf der linken Seite) explizit mit einer Cast-Anweisung (erzwungene Typumwandlung) der Typ des rechten Adress-Wertes umgewandelt werden.

- (f) Unter den in Punkten (c), (d) und (e) erläuterten Voraussetzungen sind die Operatoren `++`, `--`, `+=` und `-=` für Adress-Variablen erlaubt (insbes. also nicht für `void *`-Adress-Variablen!).

Etwa:

`p++;` abkürzend für `p = p + 1;`
`p += 5;` abkürzend für `p = p + 5;`. (Der zweite Operand ist ganzzahlig.)
`p -= a;` abkürzend für `p = p - a;`. (Der zweite Operand ist ganzzahlig.)

- (g) Sonst sind keine arithmetischen Operationen mit Zeigern zugelassen.

Wie ebenfalls bereits in Abschnitt 6.7 erwähnt, hat ein Feldname als Wert die Adresse des Feldanfangs. Mit dieser Adresse kann wie mit jedem sonstigen Adress-Wert verfahren werden.

Ist z.B. `int a[100]` die Definition eines Feldes, so hat `a` die Adresse des Feldanfangs als Wert und kann als konstanter Adress-Wert aufgefasst werden. So ist etwa `*a` synonym zu `a[0]`, `*(a+1)` synonym zu `a[1]` usw..

Umgekehrt kann ein Adress-Wert wie ein Feldname verwendet werden, s.o..

Obwohl somit Feldnamen und Adress-Variablen in vieler Hinsicht gleichartig verwendet werden können, muss jedoch ein Unterschied zwischen Feldnamen und Adress-Variablen beachtet werden: Feldnamen haben einen konstanten Wert, während der Wert von Adress-Variablen verändert werden kann.

13.2 Adressen von Funktionen

Funktionen in C sind keine Variablen, haben aber trotzdem in gewissem Sinne eine Adresse. Diese Adresse kann von einer entsprechend zu definierenden Adress-Variablen (Zeiger auf Funktion, Funktionszeiger) als Wert aufgenommen werden.

```
int (*comp) (void *, void *);
```

ist beispielsweise die Definition einer Adress-Variablen mit dem Namen `comp`, die als Wert die Adresse einer Funktion vom Rückgabotyp `int` mit zwei Zeigern auf `void` als Argumente aufnehmen kann. Der Zeiger `comp` ist wiederum typgebunden, d.h. er darf nur auf Funktionen dieses Types (oder auf `NULL`) zeigen.

(Man beachte, dass die Klammern in `(*comp)` wichtig sind, da die Zeile sonst eine Deklaration einer Funktion mit dem Namen `comp` wäre, welche zwei `void`-Zeiger als Argumente erwartet und einen Zeiger auf `int` als Wert zurückgibt!)

Der Name einer Funktion hat als Wert die „Adresse“ der Funktion und kann einem entsprechenden Funktionszeiger als Wert zugewiesen werden.

Ist etwa `fkt` eine Funktion mit zwei `void`-Zeigern als Parametern und `int`-Rückgabe (d.h. die Deklaration von `fkt` lautet: `int fkt( void *, void *);`), so verweist nach der Zuweisung

```
comp = fkt;
```

der Funktionszeiger `comp` auf die Funktion `fkt` und `*comp` ist diese Funktion. Man kann somit `(*comp)(p1,p2)` schreiben und hat damit die Funktion `fkt` mit den Argumenten `p1` und `p2` aufgerufen. Auch hier sind die Klammern in `(*comp)` wichtig, da die Priorität der Klammern von `(p1,p2)` höher ist als die von `*` und in `*comp(p1,p2)` versucht würde, den Verweis-Operator `*` auf das Objekt `comp(p1,p2)` anzuwenden!

Der Wert von Funktionszeigern kann einer erzwungenen Typumwandlung unterworfen werden. Ist etwa die Funktion `numcmp` wie folgt deklariert:

```
int numcmp(char *, char *);
```

so erwartet sie zwei `char`-Zeiger als Argumente und ihre Adresse kann somit nicht dem Funktionszeiger `comp` zugewiesen werden, da sein Funktionstyp zwei `void`-Zeiger als Argumente benötigt.

Durch die Typumwandlung in

```
comp = (int (*)(void *, void *)) numcmp;
```

wird die Funktion `numcmp` entsprechend aufgefasst und die Zuweisung ist so zulässig.

Benötigt wird dies i. Allg. immer dann, wenn einer C-Funktion eine andere C-Funktion als Argument übergeben werden soll (vergleiche etwa die Funktionen `bsearch` oder `qsort` aus der Standardbibliothek).

Beispiel:

Es gibt in der numerischen Mathematik eine Reihe von Algorithmen, welche (einen Näherungswert für) das bestimmte Integral $\int_a^b f(x) dx$ berechnen. Hierbei wird intern der Integrand nur an unterschiedlichen x -Werten ausgewertet!)

Will man einen solchen Algorithmus als C-Funktion (etwa mit dem Namen `integral`) realisieren, so müssen dieser Funktion die Integrationsgrenzen `a` und `b` (`double`-Werte) sowie die zu integrierende Funktion `f` (eine reelle Funktion, in C durch eine Funktion mit `double`-Argument und `double`-Ergebnis zu realisieren) übergeben werden. Natürlich ist das Funktionsergebnis von `integral` ein `double`-Wert.

Die Funktion müsste also wie folgt deklariert (und entsprechend definiert) werden:

```
double integral ( double a, double b, double (*f)(double) );
```

Der dritte Parameter `double (*f)(double)` ist ein Funktionszeiger, der als Wert die Adresse einer Funktion mit einem `double`-Argument und einem `double`-Ergebnis aufnehmen kann. (Innerhalb der Funktionsdefinition von `integral` kann dann mit `(*f)` und passendem Argument die zu integrierende Funktion aufgerufen werden!)

Da die Funktion `double sin(double)` aus der Standardbibliothek genau von diesem Typ ist, könnte die Integral-Berechnungs-Funktion wie folgt aufgerufen werden:

```
erg = integral( 0.0, 2.0, sin);
```

Als drittes Argument ist hier der Name einer (passenden) Funktion angegeben worden und die Parametervariable `f` erhält als Wert die Adresse der angegebenen Funktion. Das Funktionsergebnis von `integral( 0.0, 2.0, sin)` dürfte dann (bei korrekter Implementierung des entsprechenden numerischen Algorithmus) ein Näherungswert zu $\int_0^2 \sin(x) dx$ sein.

Will man eine andere Funktion integrieren, etwa ein Polynom $g(x) = x^3 + x - 1$, so braucht man diese Funktion nur als C-Funktion, etwa hornerartig wie folgt, zu definieren und entsprechend zu deklarieren:

```
double g( double x)
{
    return (x*x + 1 ) * x - 1;
}
```

und man kann zu diesem Integranden ein bestimmtes Integral, etwa $\int_{-5}^3 g(x) dx$, näherungsweise durch den Aufruf:

```
erg = integral ( -5.0, 3.0, g);
berechnen.
```

Literatur

- [Ba 88] M. Banahan, *The C Book*, Addison Wesley, 1988
- [Fe 85] A. R. Feuer, *Das C-Puzzle-Buch*, Carl Hanser Verlag, 1985
- [Ke 84] A. Kelley und I. Pohl, *A Book on C*, Benjamin/Cummings Publishing, 1984
- [Ker 83] B. W. Kernighan und D. Ritchie, *Programmieren in C*, Carl Hanser, 1983
- [Ker 88] B. W. Kernighan und D. Ritchie, *The C Programming Language*, Carl Hanser, 1988
- [Ker 90] B. W. Kernighan und D. Ritchie, *Programmieren in C*, 2-te Auflage, Carl Hanser, 1990
- [Pl 85] Th. Plum, *Das C-Lernbuch*, Carl Hanser, 1985
- [Str 87] B. Stroustrup, *Die C++ Programmiersprache*, Addison Wesley, 1987

Index

- ! Negation, 74, **139**, 204
 - != ungleich, 73, **139**, 204
 - #
 - in Makros, 161
 - leere Präprozessoranweisung, 165
 - Präprozessor, 44, **159**
 - ## in Makros, 161
 - #define, 88, **159**
 - #elif, 162
 - #else, 162
 - #endif, 162
 - #error, 164
 - #if, 162
 - #ifdef, 163
 - #ifndef, 163
 - #include, 44, 129, **159**
 - "...", 159
 - <...>, 159
 - #line, 164
 - #pragma, 165
 - %
 - in Formatstrings, 55, **167**, **170**
 - Modulo-Operator, 54, **139**
 - %%
 - in Formatstrings, 167
 - %= Zuweisung, 141
 - &
 - Adress-Operator, 57, 114ff, 117, 142
 - bitweises *UND*, 140
 - && logisches *UND*, 74, **139**
 - &= Zuweisung, 141
 - () Funktionsaufruf, 142
 - (*Typ*) Cast-Operator, 142
 - *
 - Definition von Adress-Variablen, 114, 117
 - in Ausgabespezifikation, 169
 - in Eingabespezifikation, 61, 94, **170**
 - Multiplikation, 54, **139**
 - UNIX, *siehe* Dateinamensexpansion
 - Verweis-Operator, 115ff, 118, 142
 - */ Kommentarende, 44
 - *= Zuweisung, 141
 - +
 - Addition, 54, **139**, 204
 - Vorzeichen, 139
 - ++ Inkrement, **140**, 206
 - += Zuweisung, **141**, 206
 - , Operator, 143
 - - Subtraktion, 54, **139**, 204, 205
 - Vorzeichen, 139
 - Dekrement, **140**, 206
 - = Zuweisung, **141**, 206
 - > Komponentenzugriff, 142, **196**
 - ~ Komplement, 140
 - . Komponentenzugriff, 142, **193**
 - /
 - Division, 54, **139**
 - Wurzelverzeichnis, 8
 - /* Kommentaranfang, 44
 - /= Zuweisung, 141
 - /dev/null, 16
 - <
 - kleiner, 73, **139**, 204
 - UNIX, *siehe* Eingabeumlenkung
 - <<
 - Shift-Operator, 140
 - <<= Zuweisung, 141
 - <= kleiner gleich, 73, **139**, 204
 - <bs> , *siehe* Backspace-Taste
 - <cr> , *siehe* Return-Taste
 - <ctrl> , *siehe* Control-Taste
 - , *siehe* Delete-Taste
 - <esc> , *siehe* Escape-Taste
 - <tab> , *siehe* Tabulator-Taste
- = Zuweisung, 54, 84, **141**, 194, 206
 - == gleich, 73, **139**, 204
 - >
 - größer, 73, **139**, 204
 - UNIX, *siehe* Ausgabeumlenkung

- >= größer gleich, 73, **139**, 204
- >>
 - Shift-Operator, 140
 - UNIX, *siehe* Ausgabeumlenkung
- >>= Zuweisung, 141
- >| UNIX, 15
- ?
 - UNIX, *siehe* Dateinamensexpansion
- ?: Operator, 143
- [...] Zeichenbereich, 93
- [] Feldindizierung, 142
- \ Folgezeile, 42, 159
- \ " Sonderzeichen, 43
- \0 Stringendezeichen, 90
- \\ Sonderzeichen, 43
- \n Zeilenvorschub, 42, 58
- \t Tabulator, 43
- ^= Zuweisung, 141
- _IOFBF, 174
- _IOLBF, 174
- _IONBF, 174
- __DATE__, 164
- __FILE__, 164, 186
- __LINE__, 164, 186
- __STDC__, 164
- __TIME__, 164
- | bitweises *ODER*, 140
- |= Zuweisung, 141

- a2ps, 18
- Ablaufkontrolle, 73–84
- abort, 178
- abs, 179
- abweisende Schleife, 81
- acos, 182
- Adress
 - Arithmetik, 204ff
 - !, 204
 - !=, 204
 - +, 204
 - ++, 206
 - +=, 206
 - , 204, 205
 - , 206
 - =, 206
 - <, 204
 - <=, 204
 - =, 206
 - ==, 204
 - >, 204
 - >=, 204
 - Subtraktion von Adress-Werten, 205
 - Adresse
 - als Feldname, 121
 - als Funktionsergebnis, 123
 - einer Adress-Variablen, 119
 - einer Struktur, 195
 - einer Variablen, 113
 - eines Feldes, 120
 - eines konstanten Strings, 122
 - NULL, 115
 - Typ, 113
 - typlose, 118
 - Adress-Operator, 114ff, 117, 142
 - Adress-Variable, 111, *siehe* Zeiger
 - Definition, 114
 - Alignment, 200
 - ANSI-C, 23
 - Anweisung, 39ff, **146**
 - Ausdrucks, 146
 - Auswahl, 73ff
 - break, 79–81
 - continue, 79–81
 - do-while, 81
 - for, 78
 - gelabelte, 82
 - goto, 84, 147
 - if, 73
 - if-else, 75
 - leere, 78, 91, 146
 - markierte, 147
 - Präprozessor, 44, 88
 - return, 102ff
 - Schleifen, 77–81
 - switch, 81
 - Verbund, 75, 147

- `while`, 80
 - Wiederholungs, 77ff
- Anweisungsteil
 - im Speicher, 39
- Arbeitsspeicher, 34
- Argumente der Kommandozeile, 130
- Array, *siehe* Feld
- ASCII-Zeichensatz, 67
 - Steuerzeichen, 69
 - Tabelle, 69
- `asctime`, 189
- `asin`, 182
- Assemblersprache, 22
- `assert`, 186
- `assert.h`, 186
- Assoziativität, 46, 138
- `atan`, 182
- `atan2`, 182
- `atexit`, 178
- `atof`, 178
- `atoi`, 178
- `atol`, 178
- Aufzählungskonstante, 132
- Aufzählungstypen, 132ff
- Ausdruck, 40, 78, **144**
 - bedingter, 143
 - konstanter, 82, 88, 153, 154, 163
- Ausdrucksanweisung, 146
- Ausführungsrecht, 12
- Ausgabe
 - direkte, 176
 - Flagge, 55
 - Funktionen, 166–177
 - ganzzahliger Werte, 55
 - gepuffert, 174
 - Gleitpunktwerte, 65
 - Spezifikation, 55, 92, **168**
 - Umlenkung, 15
 - von Adress-Werten, 114
 - von Strings, 92
 - Zeichen, 71
- Auswahlanweisung, 73
- Auswertungsreihenfolge, 46, 138, 139, 144

- bei `&&`, 74
- Authentifizierung, 5
- `auto`, 148
- automatische Variablen, 148
- Backspace-Taste*, 4
- Backup, 2
- Basis, 61
- Bedienoberfläche, 5
- bedingter Ausdruck, 143
- Bedingung, 73, 78
 - komplexe, 74, 85
 - Präprozessor, 162
- Benutzer
 - Kennung, *siehe* Benutzername
 - Name, 2, 11
 - ausgeben, 11
 - Nummer, 11
 - ausgeben, 11
 - Verwaltung, 2
- Bezeichner, 40, **43**
- Bibliothek, 38
 - Standard, 166–191
- Bibliotheksfunktion, 166–191
 - `abort`, 178
 - `abs`, 179
 - `acos`, 182
 - `asctime`, 189
 - `asin`, 182
 - `assert`, 186
 - `atan`, 182
 - `atan2`, 182
 - `atexit`, 178
 - `atof`, 178
 - `atoi`, 178
 - `atol`, 178
 - `bsearch`, **180**, 207
 - `calloc`, 177
 - `ceil`, 183
 - `clearerr`, 176
 - `clock`, 188
 - `cos`, 182
 - `cosh`, 182
 - `ctime`, 189

difftime, 189
div, 179
exit, 178
exp, 182
fabs, 183
fclose, 172
feof, 175
ferror, 175
fflush, 174
fgetc, 172
fgetpos, 176
fgets, 173
floor, 183
fmod, 183
fopen, 171
fprintf, 172
fputc, 172
fputs, 172
fread, 177
free, 177
freopen, 172
frexp, 183
fscanf, 173
fseek, 176
fsetpos, 177
ftell, 176
fwrite, 177
getc, 172
getchar, 60, 71, 94, 99, **167**
getenv, 178
gets, 94, **167**
gmtime, 189
isalnum, 184
isalpha, 183
iscntrl, 183
isdigit, 183
isgraph, 184
islower, 183
isprint, 184
ispunct, 184
isspace, 184
isupper, 183
isxdigit, 184
labs, 179
ldexp, 183
ldiv, 179
localtime, 189
log, 182
log10, 182
longjmp, 186
malloc, 177
memchr, 186
memcmp, 186
memcpy, 185
memmove, 186
memset, 186
mktime, 189
modf, 183
perror, 176
pow, 100, **182**
printf, 41, 55, 65, 92, 99, 114, **167**
putc, 172
putchar, 72, 92, **167**
puts, 92, **167**
qsort, **180**, 207
raise, 188
rand, 179
realloc, 177
remove, 175
rename, 175
rewind, 176
scanf, 57, 93, 96, **169**
setbuf, 174
setjmp, 186
setvbuf, 174
signal, 187
sin, 182
sinh, 182
sprintf, 173
sqrt, 182
srand, 179
sscanf, 173
strcat, 94, **184**
strchr, 185
strcmp, 94, 99, **185**
strcpy, 94, 96, 99, **184**
strcspn, 185
strerror, 185

- strftime, 189
- strlen, 94, **185**
- strncat, 185
- strncmp, 185
- strncpy, 184
- strpbrk, 185
- strrchr, 185
- strspn, 185
- strstr, 185
- atrtod, 178
- strtok, 185
- strtol, 179
- strtoul, 179
- system, 178
- tan, 182
- tanh, 182
- time, 188
- tmpfile, 175
- tmpnam, 175
- tolower, 184
- toupper, 184
- ungetc, 173
- vfprintf, 173
- vprintf, 173
- vsprintf, 173
- Bildschirm, 34
- binär, 46
- Binäre Suche, 180
- Binärstrom, 166
- Binder, *siehe* Linker
- Bit, 34, 48
- Bitfeld, 202
- Bitoperatoren, 140
- break, 79–81, 83
- Browser, 20
- bsearch, **180**, 207
- BUFSIZ, 175
- Bus, 34
- Button, 5
- Byte, 48
 - höherwertig, 49
 - niederwertig, 49
- Call by Value*, 104, 119, 169, 196
- calloc, 177
- Caps–Lock–Taste*, 4
- case–Label, 82
- cat, 15, 17
- cc, 22
- cd, 10
- CDE, 5
- ceil, 183
- char, 49
- CHAR_BIT, 190
- CHAR_MAX, 190
- CHAR_MIN, 190
- clearerr, 176
- clock, 188
- CLOCKS_PER_SEC, 188
- clock_t, 188
- Common–Desktop–Environment*, *siehe* CDE
- Compiler, 38
 - cc, 22
 - Fehler, 23, 128
 - gcc, 22, 127
- Compound–Statement*, 75, 147
- const, 134ff
- continue, 79–81
- Control–Taste*, 4
- cos, 182
- cosh, 182
- cp, 13
- CPU, 34
- ctime, 189
- ctype.h, 183
- Cut & Paste*, 17
- Darstellung
 - ganzzahliger Werte, 48ff
 - hexadezimal, 51
 - oktal, 51
 - von Gleitpunktwerten, 62
- Datei, 8
 - ansetzen, 14, 15
 - Baum, 36
 - drucken, 18
 - einfügen, 159
 - Ende, *siehe* EOF

- Header, 43
- Include, 43
- kopieren, 13
- löschen, 13
- Namensexpansion, 17
- Öffnungsmodi, 171
- Positionierung, 176
- Source, 22
- umbenennen, 13
- verschieben, 13
- versteckte, 11, 18
- Verzeichnis, *siehe* Verzeichnis
- Zugriff auf, 171ff
- Daten, 34
- Datensicherung, 2
- Datenstrom, 166
- Datenteil
 - im Speicher, 39
- Datentyp, 40, 48–72
 - char, 49
 - double, 63
 - FILE, 171
 - float, 62
 - ganzzahlig, 48ff
 - Gleitpunkt, 61ff
 - int, 40, 49
 - long, 49
 - long double, 63
 - short, 49
 - signed, 50
 - systemabhängig
 - clock_t, 188
 - div_t, 179
 - fpos_t, 176
 - ldiv_t, 179
 - ptrdiff_t, 190
 - size_t, **143**, 174, 177, 180, 184, 189, 190
 - struct_tm, 188
 - time_t, 188
 - unsigned, 49
 - Zeichen, 67ff
- Datum, 188
- DBL_DIG, 191
- DBL_EPSILON, 191
- DBL_MANT_DIG, 191
- DBL_MAX, 191
- DBL_MAX_EXP, 191
- DBL_MIN, 191
- DBL_MIN_EXP, 191
- Definition
 - einer Funktion, 101
 - ganzzahliger Variablen, 52
 - Gleitpunktvariablen, 64
 - von Adress-Variablen, 114
 - von Strukturen, 192
 - Zeichenvariablen, 70
- Deklaration
 - einer Funktion, 72, **99**, 100, 108
 - externer Variablen, 154
- Delete-Taste, 4
- difftime, 189
- Directory, *siehe* Verzeichnis
- Diskette, 34
- div, 179
- div_t, 179
- double, 63
- do-while-Schleife, 81
- Editor, 22, 38
 - vi, 22
- Eingabe
 - direkte, 176
 - Feld, 58, 66, **170**
 - Eingabefließband, 58
 - Funktionen, 166–177
 - ganzzahliger Werte, 57
 - Gleitpunktwerte, 66
 - Modell, 57ff
 - Spezifikation, 57, 83, **170**
 - Umlenkung, 15
 - von Strings, 93
 - Zeichen, 71
- elektronische Post, 2, 21
- E-Mail, **2**, 21
 - Adresse, 2
- Ende der Eingabe, 59
- Endlosschleife, 78

- enum, 132
- EOF, 59, 71, 85, 89, 107, 169, 183
- ERANGE, 179
- Ergebnistyp einer Funktion, 99
- errno, 179
- errno.h, 179
- Escape-Taste*, 4
- exit, 178
- exp, 182
- Exponent, 61
- extern, 154

- fabs, 183
- falsch*, 73, 139
- fclose, 172
- Fehlerstatus, 107
- Feld, 87–98
 - als Funktionsargument, 119
 - Breite
 - bei `printf`, 56, 65, 92, **169**
 - bei `scanf`, 60, 94, **170**
 - Grundlagen, 87ff
 - Index, 87
 - Operator, 142
 - Initialisierung, 89
 - Länge, 87
 - implizite, 90, 97
 - mehrdimensionales, 95ff
 - als Funktionsargument, 124
 - Alternative zu, 125
 - Initialisierung, 97
 - Name
 - Adresse als, 121
 - Wert eines, 120
 - Überlauf, 87, 93
 - vom Strukturtyp, 194
 - von Adress-Cariablen, 119
 - Zeichen, 90
 - Zeiger auf, 125
- Fenster, 5
- feof, 175
- ferror, 175
- Festplatte, 34
 - Controller, 34
- fflush, 174
- fgetc, 172
- fgetpos, 176
- fgets, 173
- Fibonacci-Folge*, 88
- FILE, 171
- File, 8
- file, 14
- FILENAME_MAX, 171
- File-Server, **2**, 35
- float, 62
- float.h, 190
- floor, 183
- FLT_DIG, 191
- FLT_EPSILON, 191
- FLT_MANT_DIG, 191
- FLT_MAX, 191
- FLT_MAX_EXP, 191
- FLT_MIN, 191
- FLT_MIN_EXP, 191
- FLT_RADIX, 191
- FLT_ROUNDS, 191
- fmod, 183
- Folgezeilen, 42, 159
- fopen, 171
- FOPEN_MAX, 171
- Formatstring, 55
 - % im, 55, **167**, **170**
 - %% im, 167
- for-Schleife, 78
- fpos_t, 176
- fprintf, 172
- fputc, 172
- fputs, 172
- fread, 177
- free, 177
- freopen, 172
- frexp, 183
- fscanf, 173
- fseek, 176
- fsetpos, 177
- ftell, 176
- Funktion, 99–129
 - Argument, 102, 104

- Feld, 119, 124
 - Struktur, 196
- Aufruf, 41, **99**, 102, 142
- Bekanntheit, 108
- Call by Value*, 104, 196
- Datenfluss, 105
- Datum und Uhrzeit, 188
- Definition, 101
- Deklaration, **99**, 100, 108
- Ergebnis
 - von `main`, 130
- Ergebnistyp, **99**, 107
 - Adresse, 123
- Grundlagen, 99ff
- mathematische, 181
- Name, 99
 - ohne Argumente, 106
 - ohne Ergebnis, 107
- Parameter, 102, 104
- Rekursion, 108
- Schnittstelle, 104
- Signatur, 99
- statische, 158
- Typ, 99
 - wertverändernde, 116
- Zeiger auf, 178, 180, 187, **206**
- Zwischenspeicherung des Ergebnisses, 103, 197
- fußgesteuerte Schleife, 81
- `fwrite`, 177
- ganzzahlige Aufwertung, 54, 145
- Gateway, 3
- `gcc`, 22, 127
- `getc`, 172
- `getchar`, 60, 71, 94, 99, **167**
- `getenv`, 178
- `gets`, 94, **167**
- GID, *siehe* Gruppennummer
- Gleitpunktdarstellung, 61
 - normalisiert, 61
- globale Sprünge, 186
- `gmtime`, 189
- `goto`, 84, 147
- Gruppe, 11
 - Name, 11
 - ausgeben, 11
 - Nummer, 11
 - ausgeben, 11
- Hauptanzeige, 5
- `head`, 15
- Headerdatei, *siehe* Include-Datei
- Heap*, 148
- Heimatverzeichnis, 9
- Hexadezimaldarstellung
 - ganzzahliger Konstanten, 51
 - von Zeichen-Konstanten, 70
- Homepage*, 20
- Home-Verzeichnis, *siehe* Heimatverzeichnis
- `http`, 20
- `HUGE_VAL`, 178
- Hyper-Link*, 20
- Hypertext*, 20
- `id`, 11
- Identifizier, 43
- Identifizierung, 5
- `if`-Anweisung, 73
- `if-else`-Anweisung, 75
- Include-Datei, 43, 101, 108, 133, 134, **159**
 - `assert.h`, 186
 - `ctype.h`, 183
 - eigene, 129
 - `errno.h`, 179
 - `float.h`, 190
 - `limits.h`, 190
 - `math.h`, 89, 99, **181**
 - `setjmp.h`, 186
 - `signal.h`, 187
 - `stdarg.h`, 180
 - `stddef.h`, 143
 - `stddef.h`, 190
 - `stdio.h`, 43, 99, **166**
 - `stdlib.h`, 177
 - `string.h`, 94, 99, **184**

- `time.h`, 188
- Index, *siehe* Feldindex
- Initialisierung, 78
 - einer Union, 202
 - explizit, 53
 - mehrdimensionaler Felder, 97
 - von Adress-Variablen, 115
 - von Aufzählungskonstanten, 133
 - von Feldern, 89
 - von Strukturen, 194
 - von Zeichenfeldern, 91
- Inkrementierung, 78
- Instruktion, 34, 39ff
- Instruktionsteil, *siehe* Anweisungsteil
- `int`, 40, 49
- integral promotion*, 54, 145
- Internet, 1, 20ff
 - Name, 1
- Interrupt, 187
- `INT_MAX`, 190
- `INT_MIN`, 190
- IP-Adresse, 3
- `isalnum`, 184
- `isalpha`, 183
- `iscntrl`, 183
- `isdigit`, 183
- `isgraph`, 184
- `islower`, 183
- `isprint`, 184
- `ispunct`, 184
- `isspace`, 184
- `isupper`, 183
- `isxdigit`, 184
-
- `jmp_buf`, 186
-
- Kennung, *siehe* Benutzername
- Kernighan*, 23
- Kommando, 6
 - Argumente, 7
 - Interpreter, 6
 - Optionen, 7
 - Syntax, 6
 - Zeile, 130
-
- Komma-Operator, 143
- Kommentar, 44
 - Anfang `/*`, 44
 - Ende `*/`, 44
- Komplement
 - 2-er, 50
- Komponenten einer Struktur, 192
- Konstante, 43
 - Aufzählungs, 132
 - ganzzahlig, 40, **52**
 - hexadezimal, 51
 - oktal, 51
 - Gleitpunkt, 63
 - symbolische, 71, 88, **159**
 - Zeichen, 69
 - hexadezimal, 70
 - oktal, 70
 - Zeichenkette, 41, 92
 - Wert einer, 122
- Kontrollstrukturen, 73–84
- kopfgesteuerte Schleife, 81
- Kopfzeile, 39
-
- Label, 147
- `labs`, 179
- Lader, *siehe* Loader
- `ldexp`, 183
- `ldiv`, 179
- `ldiv_t`, 179, 180
- Leserecht, 12
- Library, *siehe* Bibliothek
- `limits.h`, 190
- Linker, 38
 - Fehler, 128
- LINUX, 3
- Loader, 38
- `localtime`, 189
- `log`, 182
- `log10`, 182
- Login-Verzeichnis, *siehe* Heimatverzeichnis
- Logische Operatoren, 74
- `long`, 49
- `long double`, 63

- longjmp, 186
- LONG_MAX, 179, 190
- LONG_MIN, 179, 190
- lp, 18
- lpr, 18
- ls, 11
- L_tmpnam, 175
- l-value*, 40, 55, 140
- Mail-Server, **2**, 35
- main, 39
 - Argumente von, 130, 178
 - Funktionsergebnis von, 130
- Makros, 159
 - mit Argumenten, 160
 - vordefinierte, 163
- malloc, 177
- Mantisse, 61
- Marke, 147
- markierte Anweisung, 147
- Maschinen
 - Sprache, 22
 - Zeichensatz, 67
- Maskieren des Zeilenendes, 42, 159
- math.h, 89, 99, **181**
- mathematische Funktion, 181
- Matrix, 96
- Mauszeiger, 5
- memchr, 186
- memcmp, 186
- memcpy, 185
- memmove, 186
- memset, 186
- Metazeichen, 17
- mkdir, 12
- mktime, 189
- Modell eines Rechners, 1
- modf, 183
- Modularisierung, 126
- Modulo-Operator, 54
- more, 15
- mv, 13
- Nameserver, 3
- NDEBUG, 186
- netscape, *siehe Netscape-Navigator*
- Netscape-Navigator, 20
- Network-File-System, 9
- Netzwerk-Schnittstelle, 34
- NFS, 9
- NIS-Server, 3
- NULL, 115, 131, 204
- Nullzeichen, 90
- Numerik-Block, 4
- Objektdatei, 38
- Oktaldarstellung
 - ganzzahliger Konstanten, 51
 - von Zeichen-Konstanten, 70
- Operator, 40, 44, **46**, 138–144
 - !, 74, **139**, 204
 - !=, 73, **139**, 204
 - %, 141
 - &, 57, 114ff, 117, 140, 142
 - &&, 74, **139**
 - &=, 141
 - (), 142
 - (Typ), 142
 - *, 54, 115ff, 118, **139**, 142
 - *=, 141
 - +, 54, **139**, 204
 - ++, **140**, 206
 - +=, **141**, 206
 - ., 143
 - , 54, **139**, 204, 205
 - , **140**, 206
 - =, **141**, 206
 - >, 142, **196**
 - %, 54, **139**
 - ~, 140
 - ., 142, **193**
 - /, 54, **139**
 - /=, 141
 - <, 73, **139**, 204
 - <<, 140
 - <<=, 141
 - <=, 73, **139**, 204
 - =, 54, 84, **141**, 206

- `==`, 73, **139**, 204
- `>`, 73, **139**, 204
- `>=`, 73, **139**, 204
- `>>`, 140
- `>>=`, 141
- `?:`, 143
- `[]`, 142
- `^=`, 141
- `|`, 140
- `|=`, 141
- arithmetischer, 139
- Assoziativität, 46, 138
- binärer, 46
- bitweise, 140
- logischer, 74, **139**
- Modulo, 54
- Priorität, 46, 138
- relationaler, 73, **139**
- `sizeof`, 142
- unärer, 46
- Vergleichs, 73, **139**
- Zuweisungen, 141
- Ordner, *siehe* Verzeichnis
- Overflow, 54, 63, 178
- Parameter einer Funktion, 102
- Passwort, 2
 - ändern, 8
- `perror`, 176
- Pfad
 - absoluter, 9
 - relativer, 9
- Postfix, 140
- `pow`, 100, **182**
- Präfix, 140
- Präprozessor, 159–165
 - `#`, 165
 - `#define`, 88, **159**
 - `#elif`, 162
 - `#else`, 162
 - `#endif`, 162
 - `#error`, 164
 - `#if`, 162
 - `#ifdef`, 163
 - `#ifndef`, 163
 - `#include`, 44, **159**
 - `#line`, 164
 - `#pragma`, 165
 - Anweisung, 44, 88, **159**
 - symbolische Konstante, 71, 88, **159**
- Präzision, 56, 65, 92, 169
- Praktikumsrechner, **1**, 35
- Primärspeicher, 34
- `printf`, 41, 55, 65, 92, 99, 114, **167**
- Priorität, 46, 138
- Programmerstellung, 22ff
- Programmfluss, 41
- Programmiersprache, 22
 - Generation, 22
 - höhere, 22
 - problemorientierte, 22
- Programmkopf, *siehe* Kopfzeile
- Programmschluss, 39
- Prompt, 6
- Protokoll, 20
- Prozessor, 34
- `ptrdiff_t`, 190
- `putc`, 172
- `putchar`, 72, 92, **167**
- `puts`, 92, **167**
- `pwd`, 10
- `qsort`, **180**, 207
- Quell
 - Datei, 22
 - Programm, 22
 - Text, 22
- Quicksort, 180
- `raise`, 188
- RAM, 34
- `rand`, 179
- `RAND_MAX`, 179
- `realloc`, 177
- Rechnermodell, 1
- Rechte
 - Ausführungs, 12
 - Lese, 12

- Schreib, 12
- Vektor, 37
- register, 151
- Rekursion, 108
- remove, 175
- rename, 175
- return, 40, 102ff
- Return-Taste*, 4
- rewind, 176
- Ritchie*, 23
- rm, 13
- rmdir, 12
- Rückgabety, *siehe* Ergebnistyp

- scanf, 57, 93, 96, **169**
- SCHAR_MAX, 190
- SCHAR_MIN, 190
- Schleife, 77–81
 - do-while, 81
 - for, 78
 - while, 80
 - Zähl, 77
- Schlüsselwort, 40, 43, 45
- Schnittstelle
 - einer Funktion, 104
- Schreibrecht, 12
- SEEK_CUR, 176
- SEEK_END, 176
- SEEK_SET, 176
- Seiteneffekt, 75, 140, 141, 143, 144, 160
- Sekundärspeicher, 34
- selbstdefinierte Typen, 133
- setbuf, 174
- setjmp, 186
- setjmp.h, 186
- setvbuf, 174
- Shell, 6
- Shift-Taste*, 4
- short, 49
- SHRT_MAX, 190
- SHRT_MIN, 190
- SIGABRT, 188
- SIG_ERR, 187
- SIGFPE, 188
- SIGILL, 188
- SIGINT, 188
- signal, 187
- signal.h, 187
- Signale, 187
- Signatur einer Funktion, 99
- signed, 50
- SIGSEGV, 188
- SIGTERM, 188
- sin, 182
- sinh, 182
- sizeof, 142
- size_t, **143**, 174, 177, 180, 184, 189, 190
- Sonderzeichen, 42, 70
- Sortieren, 180
- Source-Datei, 22
- Spaltenindex, 95
- Speichergrößen, 48
- Speicherklassen, 148–158
 - von Funktionen, 158
 - von Variablen, 148ff
- Speicherlücken, *siehe* Alignment
- Speicherverwaltung
 - dynamische, 177ff
- sprintf, 173
- sqrt, 182
- srand, 179
- sscanf, 173
- Stack*, 148
- Standard
 - Ausgabe, 15, 166, 172
 - Bibliothek, 41, 166–191
 - Eingabe, 15, 166, 172
 - Fehlerausgabe, 15, 166, 172
- static, 152, 156, 158
- statischer Speicher, 148
- stdarg.h, 180
- stddef.h, 143, **190**
- stderr, 15, 172
- stdin, 15, 172
- stdio.h, 43, 99, **166**
- stdlib.h, 177
- stdout, 15, 172

- Steuerzeichen, 70, 168
- strcat, 94, **184**
- strchr, 185
- strcmp, 94, 99, **185**
- strcpy, 94, 96, 99, **184**
- strcspn, 185
- strerror, 185
- strftime, 189
- String, 90ff
 - Ausgabe, 92
 - Eingabe, 93
- string.h, 94, 99, **184**
- Stringendezeichen, 90
- strlen, 94, **185**
- strncat, 185
- strncmp, 185
- strncpy, 184
- strpbrk, 185
- strrchr, 185
- strspn, 185
- strstr, 185
- strtod, 178
- strtok, 185
- strtol, 179
- strtoul, 179
- struct, 192
- struct_tm, 188
- Struktogramm, 40
 - Element
 - do-while-Schleife, 81
 - Fallunterscheidung, mehrere Fälle, 77
 - Funktionsaufruf, 42
 - if-else, 76
 - Rücksprung, 41
 - switch, 83
 - while, 80
 - Zählschleife, 79
- Struktur, 192–203
 - Adresse, 195
 - als Funktionsargument, 196
 - als Funktionsergebnis, 197
 - Definition, 192
 - Feld, 194
 - Initialisierung, 194
 - Komponenten, 192
 - vom Strukturtyp, 199
 - rekursive, 201
 - selbstreferierend, 201
 - Variablen, 193
 - Zuweisung, 194
- Suchen, 180
- SUN-Tastatur, 3
- switch-Anweisung, 81
- symbolische Konstante, 71, 88, **159**
- Syntaxregeln, 22
- system, 178
- System-Bus, 34
- systemabhängige Größen, 190
- Tabulator, 43
 - Tabulator-Taste*, 4
- tail, 15
- tan, 182
- tanh, 182
- Tastatur, **3**, 34
 - Backspace-Taste*, 4
 - Caps-Lock-Taste*, 4
 - Control-Taste*, 4
 - Delete-Taste*, 4
 - Escape-Taste*, 4
 - Return-Taste*, 4
 - Shift-Taste*, 4
 - Tabulator-Taste*, 4
- Terminal-Controller, 34
- Terminalfenster, 6
- Tertiärspeicher, 2
- Textstrom, 166
- time, 188
- time.h, 188
- time_t, 188
- tmpfile, 175
- TMP_MAX, 175
- tmpnam, 175
- Token, 44, 89, 159
- tolower, 184
- toupper, 184
- Typ, *siehe* auch Datentyp

- Aufzählungs, 132
- einer Adresse, 113
- einer Funktion, 99
- einer Variablen, 112
- selbstdefinierter, 133
- Struktur, 192
- Umwandlung, 54ff, 100, 102, 104, 143, 145ff
 - erzwungene, 142
- typedef**, 133, 143
- Typumwandlung, 178
- Übersetzungseinheiten, 126
- UCHAR_MAX, 190
- Überlauf, 54
- Übersetzungsfehler, 23
- Uhrzeit, 188
- UID, *siehe* Benutzernummer
- UINT_MAX, 190
- ULONG_MAX, 179, 190
- Umwandlungsspezifikation
 - zu **printf**, 167ff
 - zu **scanf**, 169ff
- Umwandlungszeichen, 55
 - zu **printf**, 168
 - zu **scanf**, 170
- unär, 46
- underflow, 63
- ungetc**, 173
- Union, 201
 - Initialisierung, 202
- union**, 201
- UNIX, 1, 5ff
 - Abmelden, 5
 - Anmelden, 5
 - Dateisystem, 8–10
 - Ein/Ausgabeumlenkung, 15ff
 - Grundlagen, 5–19
 - Kommando, 7
 - a2ps**, 18
 - cat**, 15, 17
 - cd**, 10
 - cp**, 13
 - file**, 14
 - head**, 15
 - id**, 11
 - lp**, 18
 - lpr**, 18
 - ls**, 11
 - mkdir**, 12
 - more**, 15
 - mv**, 13
 - pwd**, 10
 - rm**, 13
 - rmdir**, 12
 - tail**, 15
 - unsigned**, 49
 - USHRT_MAX, 190
 - va_arg**, 181
 - va_end**, 181
 - va_list**, 180
 - va_start**, 180
 - Variable Argumentliste, 180
 - Variablen, 40
 - Adress, 111ff
 - Adresse, 113
 - auto**, 148
 - automatische, 148
 - Definition, 39ff
 - extern**, 154
 - externe, 154
 - globale, 156
 - lokale, 156
 - Merkmale, 112
 - Name, 112
 - register**, 151
 - Speicherbereich, 112
 - static**, 152, 156
 - statische externe, 156
 - statische interne, 152
 - Typ, 112
 - vom Aufzählungstyp, 132
 - vom Strukturtyp, 193
 - Wert, 112
 - Vaterverzeichnis, 8
 - Vektor, *siehe* Feld
 - Verbundanweisung, 75, 147

- Vergleichsoperatoren, 73
- Verweis-Operator, 115ff, 118, 142
- Verzeichnis, 8
 - aktuelles, 9, 10
 - Pfad ausgeben, 10
 - erzeugen, 12
 - Heimat, 9
 - Inhalt ausgeben, 11
 - löschen, 12
 - umbenennen, 13
 - verschieben, 13
 - verstecktes, 11, 18
 - wechseln, 10
- `vfprintf`, 173
- `vi`-Editor, 22, 24–33
 - Befehlswiederholung, 32
 - Cursorpositionierung, 26
 - Dateioperationen, 31
 - Einfügemodus, 25
 - `ex`-Modus, 25
 - Kommandomodus, 24
 - Löschen, 27
 - Modi, 24
 - Pufferoperationen, 30
 - reguläre Ausdrücke, 28
 - Rückgängigmachen von Befehlen, 33
 - Sucheinstellungen, 29
 - Suchen, 28
 - Suchen und Ersetzen, 29
 - System-Kommandos, 32
- virtueller Bildschirm, 5
- `void`, 106, 107, 118
- `volatile`, 137
- Vorrangstufe, 46
- `vprintf`, 173
- `vsprintf`, 173
- wahr*, 73, 139
- Wahrheitswert, 73, 139
- Wert
 - einer Variablen, 112
- `while`-Schleife, 80
- Wiederholungsanweisungen, 77–81
- World-Wide-Web*, *siehe* *WWW*
- Wurzelverzeichnis, 8, 9
- WWW*, 20
 - Browser, 20
 - Server, 3, 20
- Zählschleife, 77
- Zeichen, 67ff
 - Bereich, 93
 - Bibliotheksfunktionen, 183
 - erlaubte, 45
 - Rechnen mit, 70
- Zeichenfeld, 90ff
- Zeichenketten
 - Bibliotheksfunktionen, 184
 - konstante
 - adjazente, 161
 - konstante, 41, 92
 - adjazente, 43
- Zeiger, *siehe* Adress-Variable
- Arithmetik, *siehe* Adress-Arithmetik
- auf Felder, 125
- auf Funktionen, 178, 180, 187, **206**
- auf `const`, 135
 - als Funktionsparameter, 136
- `const`, 135
- vom Strukturtyp, 195
- Zeilenindex, 95
- Zeilenvorschub, 42, 58, 166
- Zufallszahl, 179
- Zwischenraumzeichen, 170
- Zwischenspeicherung
 - von Funktionsergebnissen, 103, 197