0 ist false, alles andere true

Eine Einführung in
die Programmierung mit *ANSI-C*

Klaus Wachtler

31. Oktober 2003

Inhaltsverzeichnis

1 Einführung

Dieses Buch ist einerseits als Arbeitsunterlage für *C*-Kurse oder für das Selbststudium und andererseits als Nachschlagewerk für die praktische Arbeit gedacht.

Form und Inhalt sind:

Breidingstr. 17
29614 Soltau
Tel.: 0̄5191/70271
email: AnyWare@Wachtler.de

Für weitere Kopien des Buches im jeweils aktuellen Stand bitte bei obiger Adresse nachfragen.

Am Ende des Buches befindet sich ein Stichwortverzeichnis.

Der Text wurde mit TeX und LaTeX gesetzt.

Voraussetzungen:

Vorausgesetzt wird zumindest eine grobe Vorstellung der folgenden Begriffe:

- Hardware, Software
- Prozessor, Speicher, Register, Daten, Adressen
- Bit, Byte
- Betriebssystem, Programm
- Programmieren: Quelltext, Maschinensprache, Übersetzen (Kompilieren)
- Massenspeicher, Datei
- Daten, Variable, Konstante
- Deklaration, Definition

Wenn diese Voraussetzungen fehlen, ist es sinnvoll zuerst weitere Grundlagenliteratur zu konsultieren, beispielsweise [KW: EinfInf].

Die Sprache *C* ist als erste Programmiersprache völlig ungeeignet. Die Kenntnis zumindest einer weiteren Programmiersprache wird in diesem Kurs angenommen. Idealerweise ist dies *Pascal*, da diese Sprache als erste Programmiersprache sehr gut geeignet und vom Aufbau her vergleichbar ist. Der praktische Umgang mit einer anderen verbreiteten Sprache wie *FORTRAN* ist aber als Grundlage natürlich ebenso brauchbar. Von *BASIC* wird zum Einstieg dringend abgeraten.

Literatur: Mit Hilfe dieses Buches sollte zusätzliche Literatur zur Sprache *C* selbst nicht nötig sein.

Für reine Lehrzwecke sowie für die meisten praktischen Fälle sollte dieser Text reichen, soweit es nur um den eigentlichen Sprachumfang von *C* geht. Auf die vielfältigen Besonderheiten der verschiedenen Betriebssysteme oder spezieller Implementationen wird hier nicht eingegangen; dieser Text behandelt lediglich den genormten Sprachumfang von *C*. Dementsprechend steht hier auch nirgends, wie man mit dem jeweiligen System ein *C*-Programm erstellt, übersetzt und startet.

Obwohl versucht wurde, den als *ANSI-C* genormten Sprachumfang hier möglichst präzise darzustellen, ist dies in einem reinen Einführungskurs leider nicht an allen Stellen möglich.

In Zweifelsfällen ist deshalb der *C*-Klassiker (der „K&R") die letzte Instanz. Die erste Auflage [K&R I] beschreibt das veraltete, sogenannte K&R-*C*. Die deutsche Übersetzung davon ist erhältlich, aber unbrauchbar.

Aktueller —nämlich über *ANSI-C*— ist natürlich die 2. Ausgabe [K&R II]; dieses Buch ist auch in der deutschen Ausgabe empfehlenswert.

Die Sprachbeschreibung allein (in diesem Buch oder in [K&R II]) reicht zur praktischen Arbeit noch nicht aus; der Umgang mit dem jeweils verwendeten Rechnersystem und den dortigen Dienstprogrammen (Editor, Compiler etc.) ist in weiteren Handbüchern zu erlernen. Kennt der Leser aber schon ein Entwicklungssystem auf dem entsprechenden Rechner, dann wird der Umstieg auf das entsprechende *C*-System von der eigentlichen Handhabung her keine Probleme bereiten.

Der Titel wird in Abschnitt **8.4.2 Logischer Datentyp(*boolean, logical*)** auf Seite 39 verständlich; siehe dazu bei Gelegenheit die Fußnote auf Seite 39.

Zur Schreibweise: Soweit es die Verständlichkeit fördert, wurden deutsche Begriffe verwendet. Dies ist für eine Einführung mit Sicherheit besser, als mit englischen bzw. amerikanischen Schlagwörtern den Leser zu verschrecken. Andererseits sind viele gebräuchliche englische Begriffe so prägnant und verbreitet, daß eine erzwungene Übersetzung nur verwirrt.

Deshalb werden hier von Fall zu Fall beide Wege verfolgt.

Variablennamen, Programmbeispiele, *C*-Schlüsselworte etc. sind `in dieser Schrift` gehalten. In dieser Schrift werden Dateinamen geschrieben, wie beispielsweise stdio.h. Wo nötig, werden Zeichen oder Zeichenfolgen aus *C* vom umgebenden Text abgesetzt, wie etwa ein Semikolon `;`. Gerade wenn ein laufender Text mit vielen Variablennamen (wie beispielsweise `a`, `b` und `c`) und Sonderzeichen (`;`,`:`,`+`,`-`,`*`) aus *C* durchsetzt ist, finde ich diese Form wesentlich lesbarer. Funktionsnamen scheinen den Lesefluß nicht so sehr zu stören.

Ich würde dazu aber gerne Meinungen oder Vorschläge von anderer Seite hören.

Warenzeichen/Marken: Im laufenden Text werden Warenzeichen, Marken oder ähnliche geschützte Begriffe nicht laufend als solche gekennzeichnet. Trotzdem will ich mir deren Inhalt nicht böswillig zu eigen machen.

Einige (aber sicher nicht alle) verwendeten derartigen Begriffe:

- **Linux** ist ein eingetragenes Warenzeichen von Linus Torvalds.
- **Unix** ist ein eingetragenes Warenzeichen von The Open Group
- Die Bezeichnung **AnyWare** ist eine eingetragene Marke von Klaus Wachtler.
- **Visual Age** ist ein Warenzeichen von IBM.
- **Microsoft**, **Microsoft Windows**, **Visual C++** sind Warenzeichen von Microsoft.

2 Wozu C?

Um die Sprache *C* etwas einordnen zu können, werden einige wichtige Programmiersprachen angerissen:

Niedere Sprachen

Maschinensprache ist für Mensch und Tier unverständlich und rechnerabhängig. Jede Anweisung in Maschinensprache ergibt eine atomare Aktion des Prozessors. Zum Abarbeiten muß das Programm im Arbeitsspeicher stehen.
Aufbewahrt wird ein Programm in der Regel als Datei auf einem Massenspeicher (Programmdatei) und zum Abarbeiten wird es mit dem Lader (Teil des Betriebssytems) in den Speicher gebracht.

Assembler ist die lesbare Schreibweise der Maschinensprache.
Jeder Assemblerbefehl wird mit dem Assembler (-programm) zu einer Anweisung in Maschinensprache übersetzt.
Z.B. wird die Anweisungsfolge

```
MOVE D0,123465   * verschiebe den Inhalt des Registers D0
                 * an die Stelle 123465 im Speicher
SUB  #5,D1       * ziehe vom Inhalt des Registers D1 die
                 * Zahl 5 ab
ADD  6,D1        * addiere die Zahl, die an der Stelle
                 * 6 im Speicher
                 * steht, zum Inhalt von Register D1
```

vom Assembler in eine Folge von entsprechenden Maschinenbefehlen (als Binärzahlen) umgewandelt. Da die Befehle auf den Befehlsvorrat des verwendeten Rechners abgestimmt sind, ist Assembler ebenso rechnerabhängig wie die eigentliche Maschinensprache. Der Fortschritt liegt nur in der etwas leichteren Programmierung.

Makroassembler ermöglichen die Abkürzung von häufigen Befehlssequenzen durch sogenannte Makros. Die Makros erweitern also den Befehlsvorrat eines Prozessors scheinbar. Kann ein Prozessor beispielsweise nicht zwei Zahlen im Speicher addieren, sondern nur in Registern, dann kann man sich ein Makro etwa so schreiben:

```
MACRO ADDMEM  /1,/2   * addiert die Zahl an der Stelle /1 zur
                      * Zahl an der Stelle /2. Die Register
                      * D0 und D1 werden dabei ueberschrieben.
MOVE   /1,D0          * 1. Operanden holen
MOVE   /2,D1          * 2. Operanden holen
ADD    D0,D1          * addieren
MOVE   D1,/2          * zurueckschreiben
ENDM                  * Ende des Makros
```

und dann jederzeit in seinem Programm mit dem Befehl

```
ADDMEM 123,456        * Inhalt an der Stelle 123 zur Zahl an
                      * der Stelle 456 addieren.
```

zwei Zahlen im Speicher addieren.

Tatsächlich gibt es den Befehl `ADDMEM` für den Prozessor gar nicht; beim Assemblieren wird der Befehl `ADDMEM` durch die Befehlsfolge aus der Makrodefinition ersetzt. Die Parameter `/1` und `/2` werden dabei durch die Argumente `123` bzw. `456` ersetzt.

Maschinensprache und Assembler sind sehr fehleranfällig, unübersichtlich, und schwer zu warten (Wartung ist die meiste Arbeit an der Softwareentwicklung, nicht die Neuerstellung von Programmen!).

Hochsprachen

Um die Nachteile der Assemblerprogrammierung zu umgehen, wurden die Hochsprachen geschaffen (*FORTRAN*, *PL/1*, *ALGOL*, *BASIC*, *Cobol*, *RGB*, *Pascal*, *C*, *Modula2*, *ADA*, ...). Diese sind nicht mehr maschinenorientiert (Befehlsvorrat orientiert sich am verwendeten Prozessor), sondern anwendungsorientiert (der Befehlsvorrat orientiert sich an der Anwendung).

Die Hochsprachen sind also künstlich (unabhängig von einem bestimmten Prozessor) geschaffen, um bestimmte Anwendungen lösen zu können.

Die Befehle bestehen aus Anweisungen, um etwas auszugeben, Rechenoperationen durchzuführen, Daten einzulesen usw.. Da aber kein Prozessor solche Anweisungen direkt versteht, müssen Hochsprachenprogramme erst in Maschinensprache umgewandelt werden.

Dazu gibt es zwei Möglichkeiten:

- *Compiler* übersetzen das gesamte Hochsprachenprogramm (Quelltext), ohne das Programm auszuführen und erzeugen eine Datei mit allen erzeugten Prozessorbefehlen. Diese Datei ist in der Regel nicht direkt ein ausführbares Programm, sondern nur der Teil eines Programms, der auch im entsprechenden Quelltext steht. Fast immer ist ein Programm in Hochsprache nicht vollständig in einem Quelltext geschrieben. Vielmehr werden oft benötigte Unterprogramme getrennt übersetzt und liegen dann für immer als übersetzte Datei vor. Zusammen mit den selbstgeschriebenen Programmteilen, die ebenfalls aus mehreren Teilen bestehen können, wird vom Linker oder Binder ein ausführbares Programm erzeugt. Erst dieses kann dann in den Speicher geladen und ausgeführt werden.

- *Interpreter* dagegen übersetzen immer nur eine Anweisung des Programms und führen den dabei erzeugten Code gleich aus. Mehrfach durchlaufene Programmteile werden dabei auch mehrfach übersetzt; wird ein Programm oft ausgeführt, dann wird es auch entsprechend oft übersetzt, was den Programmlauf natürlich bremst. Außerdem sind solche Programme nie ohne Interpreter lauffähig. Zur Programmentwicklung können Interpreter aber sehr hilfreich sein.

Inzwischen gibt es (insbesondere für einige *BASIC*-Versionen, aber ganz selten auch für vernünftige Sprachen) Entwicklungssysteme, die Compiler und Interpreter beinhalten. Mit dem Interpreter kann man Programme entwickeln, und dann mit dem Compiler erst übersetzen, wenn der Code in Ordnung ist.

FORTRAN ist die älteste und (in den Versionen *FORTRAN 66* und *FORTRAN 77*) meistverbreitete Hochsprache. Sie dient hauptsächlich der Numerik (Maschinenbau, Elektrotechnik, Bauingenieurwesen, Mathematik etc.). Von der Struktur her veraltet. Reine Compilersprache. Für den zugedachten Einsatzzweck gut und effektiv; darüberhinaus beschränkte Fähigkeiten. Die Normung von *FORTRAN* ist sehr strikt, so daß auch größere Fortranprogramme in aller Regel sehr gut auf andere Rechnersysteme portiert werden können. Dies ist bei kommerzieller Programmentwicklung sehr wichtig, da die sehr zeitintensive und damit teure Entwicklung von Programmen oder Programmteilen dadurch nicht auf einen sehr kleinen Abnehmerkreis mit einem bestimmten Rechner- oder Compilersystem beschränkt ist. Für größere Rechner, die in den genannten Gebieten oft fast nur in *FORTRAN* programmiert werden, existieren sehr gute Compiler, die den jeweiligen Quelltext in ein sehr stark optimiertes Maschinenprogramm übersetzen und damit die Rechenleistung einer Anlage gut nutzen.

Die neueste Normung (*FORTRAN 90*) hat viele Erweiterungen, ist vielleicht etwas überladen und im gesamten Umfang wohl nicht sehr populär. Einige der Erweiterungen werden aber wohl zunehmend genutzt.

BASIC ist ein Abkömmling von *FORTRAN*, wird aber fast immer interpretiert.

Taugt eigentlich nur zum Lernen, und da auch noch schlecht. Schlecht genormt, uneffektive Erstellung von größeren Programmen, Quelltexte nicht portierbar.

Cobol ist eine rein kaufmännische Sprache. Compilersprache, Normung recht gut, aber sehr beschränkte Fähigkeiten und uneffektive Programmerstellung.

Algol war die erste Sprache mit einer modernen Blockstruktur und hat nur noch geschichtlichen Wert.

Pascal war ursprünglich als reine Lernsprache gedacht und ist dafür sehr gut geeignet. Inzwischen in vielen verschiedenen Erweiterungen (*Turbo-Pascal*, *Prospero-Pascal*, *UCSD-Pascal*) verbreitet. Die Erweiterungen sind meist sehr fähig, aber die dadurch enstehenden Dialekte zu unterschiedlich, um Programme übertragen zu können. Der übertragbare, genormte Teil von Pascal ist dagegen von der Funktion her so stark eingeschränkt, daß man nicht vernünftig damit arbeiten kann.

Es gibt auch eine genormte Erweiterung von Pascal, nämlich *Extended Pascal*, die aber nicht sehr verbreitet ist.

C wurde urspünglich zur Systemprogrammierung entwickelt (im Rahmen der Erstellung des Betriebssystems *Unix*), also zum Schreiben von maschinennahen Programmen wie Betriebssystemen oder Teilen davon oder Dienstprogrammen wie Compiler, Editoren etc..

Dabei sollte *C* die für diese Zwecke ausschließlich verwendete Assemblerprogrammierung möglichst weitgehend ersetzen, da die immer größeren und leistungsfähigeren Betriebssysteme in Assembler zunehmend uneffektiv zu erstellen waren.

Das Konzept der Sprache ist aber so flexibel, daß man damit praktisch alle vorkommenden Programmierprobleme lösen kann. Der Compiler selbst ist relativ klein und einfach; dadurch steht *C* auf jedem neueren Rechner schnell zur Verfügung. Die Anwendungsbreite von *C* ist wohl größer als die der meisten anderen Sprachen.

Die Anweisungen in *C* sind wie bei den anderen Hochsprachen anwendungsorientiert, aber andererseits so definiert, daß sie sehr effektiven Maschinencode nach dem Übersetzen ergeben. Zusammen mit der Möglichkeit von *C*, direkt im Speicher des Rechners Werte lesen und schreiben können, kann man sehr maschinennahe Programme verwirklichen und trotzdem den Quelltext so portabel wie möglich halten. *C* läßt dem Programmierer sehr viele Freiheiten und behindert praktisch keinen Wunsch.

Zusammen mit der sehr effektiven und schnellen Programmerstellung bietet sich dadurch *C* für kommerzielle Programmentwicklung an. Besonders nachteilig ist die Kehrseite der Freiheit: Im Gegensatz zu mehr restriktiven Sprachen wie *Pascal* oder

Modula2 erlaubt *C* nicht nur (fast) alles, was der Programmierer bezweckt, sondern auch alles, was er falsch macht. Beispielsweise kann man Rückgabewerte von Funktionen ignorieren, ohne einen *C*-Compiler nennenswert zu stören. Ein Pascalcompiler bekommt dabei einen halben Herzinfarkt. Ebenso kann man einen Zeiger auf einen bestimmten Datentyp mühelos einem Zeiger auf einen anderen Datentyp zuweisen.

Dadurch ist die Programmierung in *C* sehr fehlerträchtig, wenn man nicht freiwillig und mit viel Selbstdisziplin einige Spielregeln einhält. Noch wichtiger wird dies, wenn mehrere Leute an einem größeren Projekt arbeiten.

Eigentlich gibt es zwei verschiedene *C*:

- Einmal das alte, sogenannte *K&R-C*.

  Der Name kommt von den Herren Brian W. ***K***ernighan **&** Dennis M. ***R***itchie. Von Dennis Ritchie stammen die Entwicklung von *C* und die erste Implementation (auf einer PDP 11). Die Sprache wurde von K&R erstmals in dem Buch [K&R I] genau beschrieben. Die davon existierende deutsche Übersetzung ist für meinen Geschmack total mißlungen.

- Zum anderen das neuere *ANSI-C*[1].

  Dazu gibt es ebenfalls von K&R die zweite Ausgabe ihres Buches ([K&R II]). Davon ist auch die deutsche Übersetzung sehr gut zu lesen und korrekt.

  Die größten Vorteile von *ANSI-C* sind zum einen, daß einige unnötige Fehleranfälligkeiten beseitigt wurden und zum anderen, daß die Standardisierung wesentlich besser wurde. In dieser Sprache geschriebene Programme sind recht gut portabel, soweit auf den entsprechenden Rechnern *ANSI-C* implementiert ist. Dies ist auf praktisch allen Systemen der Fall.

  Viele existierende Programme sind noch nicht in *ANSI-C*, sondern im (alten) *K&R-C* geschrieben.

  Neuentwicklungen werden aber nur noch in *ANSI-C* stattfinden, da es für die Entwicklung von Programmen deutliche Vorteile gegenüber dem Original hat und relativ strikt genormt ist.

  In diesem Kurs wird nur *ANSI-C* behandelt.

Mit dem Aufkommen von objektorienterten Denkansätzen entstanden auch dementsprechende Weiterentwicklungen von *C*. Dabei hat ganz klar *C++* das Rennen gemacht ([Stroustrup: C++]); diese Sprache ist im wesentlichen eine Erweiterung von *C* und ist inzwischen praktisch in fast allen Bereichen der Softwareentwicklung dominierend. Insbesondere für große Entwicklungsprojekte ist diese Sprache gut geeignet.

Da *C++* den Sprachumfang von *C* fast exakt als Untermenge enthält, ist in *C++* zumindest alles möglich, was auch in *C* machbar ist: im Guten wie im Bösen. Da in [Stroustrup: C++] zwar sehr schön alles beschrieben ist, was über *C* hinausgeht, aber wenig auf die möglichen Probleme eingegangen wird, die man sich mit *C* aus Unkenntnis einhandeln kann, halte ich gar nichts davon, *C++* als Einsteigersprache zu empfehlen, ohne zuerst ein klares Verständnis zu entwickeln.

[1] Mit *ANSI-C* ist hier der Sprachumfang gemeint, der ANSI-Norm X3.159-1989 (ISO 9899-1990) definiert ist. Es gibt etliche weitere Standards, die aber nur eine beschränkte Bedeutung haben. Zum einen definieren einige Standards weitere Regeln, die nur auf bestimmten Systemen gelten (POSIX), zum anderen sind direkte Weiterentwicklungen von ANSI-C (C 99) insofern obsolet, als C in C++ aufgeht, und damit fehlende Erweiterungen dadurch nachgereicht werden.

Wenn man *C++* allerdings als Weiterentwicklung von *C* auffaßt, kann man sehr viele Vorteile aus dieser Sprache ziehen.

Andere Erweiterungen von *C* (*Objective C*) dürften sich nicht mehr in voller Breite durchsetzen.

Microsoft versucht derzeit, seine Neuschöpfung *C#* in den Markt zu drücken. Dies ist von der Grundidee eine Kreuzung zwischen *C++* und *Java*; wo der große Vorteil gegenüber diesen bereits existierenden Sprachen liegt, wird nicht so recht klar (außer daß Microsoft versucht, das verhaßte *Java* des verhaßten Konkurrenten Sun zu verdrängen). Ein technischer Fortschritt scheint es jedenfalls nicht zu sein, eher ist das Gegenteil der Fall.

Java ist inzwischen im Bereich der objektorientierten Programmierung (OOP) auch sehr weit verbreitet, und wird oft als direkte Konkurrenz zu *C++* gesehen. Diese beiden Sprachen haben aber sehr unterschiedliche Stärken und Schwächen, wodurch sich *Java* eher zur Erstellung nicht zu großer, sehr portabler Projekte eignet (insbesondere wegen der recht umfangreichen portablen Bibliotheken, beispielweise zum Erstellen der Benutzeroberfläche eines Programms), *C++* dagegen ab einer gewissen Projektgröße die Nase vorne hat und spätestens dann unentbehrlich wird, wenn auf Laufzeiteffizienz geachtet werden muß.
Java nimmt eine Sonderstellung hinsichtlich der Unterscheidung in Compiler- und Interpretersprachen ein: ein Java-Quelltext wird zwar kompiliert, aber üblicherweise nicht zu Maschinensprache, sondern in einen sogenannten Bytecode (p-Code, Pseudocode), der rechnerunabhängig ist. Dieser wird dann zur Laufzeit von einem systemabhängigen, aber vom Anwendungsprogramm unabhängigen Interpreter ausgeführt.

(Zur Einordnung sollen diese Vergleiche genügen; es gibt natürlich noch viel mehr Sprachen als die hier aufgeführten. Die nicht erwähnten haben großteils auch eine Daseinsberechtigung, ohne hier explizit erwähnt worden zu sein.)

3 Kleine Beispiele

In diesem Kapitel werden anhand von Beispielen die wichtigsten Aspekte kurz dargestellt, um kleine *C*-Programme verstehen zu können.

Ein *C*-Programm besteht aus einem oder mehreren *Quelltexten*, die getrennt voneinander übersetzt werden können und dann mit den bereits vorhandenen Routinen aus Bibliotheken zusammengebunden werden zu einem hoffentlich lauffähigen Programm.

Jeder Quelltext besteht dabei aus einer Folge von *Kommentaren*, *Praeprozessoranweisungen*, *Definitionen* und *Deklarationen*.

Nach dem Durchlauf durch den Praeprozessor besteht der *C*-Quelltext noch aus Definitionen und Deklarationen, und zwar sowohl von Daten als auch von Programmcode.

In allen Quelltexten, die übersetzt und zusammengebunden werden, muß es genau eine Funktion mit dem Namen `main()` geben. Diese Funktion spielt die Rolle des Hauptprogramms bei anderen Sprachen: mit der ersten ausführbaren Zeile in `main()` beginnt die Programmausführung, und sie endet mit der letzten ausführbaren Zeile (oder mit einer `return`-Anweisung in `main()` oder einem Programmabbruch, beispielsweise durch einen Laufzeitfehler oder einen Aufruf von `exit() oder abort()`).

Eigentlich hat `main()` zwei Parameter, die wir aber bis auf weiteres unterschlagen wollen. Wenn eine Funktion mit Parametern aufgerufen wird, die aber gar keine Parameter verwendet, ist das in *C* kein Fehler.

Solange wir also die beiden Parameter, die `main()` bei jedem Programmlauf mit auf den Lebensweg bekommt, gar nicht benötigen, tun wir so, als gäbe es sie nicht.

Letztlich steckt in den beiden Parametern die Kommandozeile, mit der das Programm nach dem Übersetzen und Linken aufgerufen wird. Die Kommandozeile interessiert uns noch nicht, also lassen wir die Parameter einfach weg.

Als weitere Vereinfachung soll unser Programm nur aus einem Quelltext bestehen.

Wie sieht nun ein Programm aus, das einen Text auf dem Bildschirm ausgibt?

Wir wollen den Text `Pause!` auf den Bildschirm schreiben.

Ein String, also eine Zeichenfolge, wird in *C* von Gänsefüßchen `"` eingeklammert.

Um einen solchen String auszugeben, gibt es bereits bei jedem *C*-System eine Funktion namens `puts()` (von: *put string*), die man nur mit dem gewünschten String aufrufen muß.

Anstandshalber sollte man jede verwendete Funktion am Anfang des Quelltextes vereinbaren. Dies ist zwar oft nicht nötig, so auch in diesem Beispiel eigentlich nicht. Aber weggelassene Funktionsdeklarationen machen oft Ärger. Deshalb ist es eine ganz hilfreiche Angewohnheit, alles zu deklarieren, was man verwendet; einschließlich aller aufgerufenen Funktionen.

Eine Funktionsdeklaration (sogenannter *Funktionsprototyp*) besteht aus dem Typ des Rückgabewerts, dem Funktionsnamen, einem runden Klammerpaar, in dem die Parameter vereinbart werden (falsl vorhanden), und einem Semikolon `;`.

Als Rückgabewert ist jeder Datentyp zulässig, oder das Schlüsselwort `void`, wenn die Funktion nichts zurückgibt. Dieser Fall entspricht einer `SUBROUTINE` in *FORTRAN* oder einer `PROCEDURE` in *Pascal*. Läßt man die Angabe des Rückgabewertes weg, dann nimmt der Compiler `int` als Rückgabewert an.

Die Parameter werden nacheinander mit Typ und eventuell einem Namen vereinbart. Den Namen kann man bei einer Deklaration einfach weglassen; er ist nur bei der Definition einer Funktion nötig. Die einzelnen Parameter werden mit Kommata getrennt. Gibt man zwischen den Klammern gar keine Parameter an, dann heißt das nicht, daß die Funktion keine Parameter erhalten darf! Vielmehr kontrolliert der Compiler dann einfach beim Aufruf der Funktion keine Datentypen. Will man eine Funktion deklarieren, die gar keine Parameter erhalten soll, dann gibt man anstelle der Parameterliste ebenfalls das Schlüsselwort `void` an. Ruft man dann die Funktion später doch mit Parametern auf, erhält man vom Compiler eine entsprechende Fehlermeldung. In *C* ist es (im Gegensatz zu fast allen anderen Sprachen) durchaus möglich und legal, einer Funktion bei jedem Aufruf eine unterschiedliche Anzahl von Parametern zu übergeben. Neben der Anzahl sind auch die Typen der Parameter variabel. In *FORTRAN* ist dies nur bei vordefinierten Funktionen möglich (z.B. `MAX`), in *Pascal* ebenso (`write`).

Ein vollständiges Programmbeispiel sieht also etwa so aus:

```
/* klpause0.c 30. 7.95 kw
 *
 * Dieses Programm gibt einen sehr wichtigen Text auf dem Bildschirm
 * aus und beendet sich dann gleich wieder.
 */

int puts( const char *s );         /* Diese Funktion wird verwendet,
                                    * also lieber vereinbaren.
                                    * puts() gibt einen String s aus
                                    * und haengt selbstaendig einen
                                    * Zeilenvorschub an.
                                    */

/* Das ist das Hauptprogramm. Die beiden Parameter von main() werden
 * ignoriert.
 */

int main()
{
  puts( "Pause!" );          /* Den gewuenschten Text ausgeben      */
  return 0;                  /* und den Wert 0 zurueckgeben.        */
}
```

`main()` ist trotz ihrer besonderen Bedeutung als Hauptprogramm eine Funktion wie jede andere auch.

Die Funktion `main()` liefert einen Wert vom Typ `int`, also eine ganze Zahl.

Was mit dem Rückgabewert der Funktion `main()` passiert, ist von der jeweiligen Implementation abhängig.

Bei den meisten Systemen (zumindest *MSDOS*, *Unix* und Verwandte der beiden) geben Programme einen Zahlenwert zurück. In aller Regel ist dies eine `0`, wenn das Programm im wesentlichen fehlerfrei lief, oder eine `1` bei leichten Fehlern (zum Beispiel Compiler, wenn sie Warnungen ausgegeben hatten). Bei schwereren Fehlern wird entsprechend ein größerer Wert zurückgegeben. Beispielsweise gibt ein Compiler eine `0` zurück, wenn der Quelltext korrekt übersetzt wurde; eine `1`, wenn beim Übersetzen Warnungen ausgegeben wurden und eine `2`, wenn so schwere Fehler auftraten, daß kein lauffähiges Kompilat erstellt werden konnte.

Solche Systeme verwenden den Rückgabewert von `main()` dann auch als Rückgabewert des Programms. In der Regel sollte man also `main()` mit `return 0;` beenden.

`main()` kann jetzt aber auch wieder andere Funktionen aufrufen. Das können bereits vorhandene Funktionen aus einer Bibliothek mit fertig übersetzten Funktionen sein (wie `puts()`), als auch selbstgeschriebene in der selben oder einer anderen Quelldatei.

Zur Vereinfachung wird die Möglichkeit, ein Programm aus mehreren Quelldateien zusammenzusetzen, für die nächsten Beispiele nicht genutzt.

Die flexibelste Möglichkeit, Text auf dem Bildschirm auszugeben, ist die Funktion `printf()`. Sie erhält immer mindestens einen Parameter, nämlich einen Zeiger auf eine Zeichenkette, die auszugeben ist.

Damit kann man das erste Beispiel auch anders schreiben:

```
/* klpaus1.c 30. 7.95 kw
 *
 * Dieses Programm gibt einen sehr wichtigen Text auf dem Bildschirm
 * aus und beendet sich dann gleich wieder.
 */

int printf( char *format, ... );   /* gibt format und eventuell
                                    * weitere Werte auf die Standard-
                                    * ausgabe.
                                    */

int main()
{
  printf( "Pause!\n" );       /* Den gewuenschten Text ausgeben      */
  return 0;                   /* und den Wert 0 zurueckgeben.        */
}
```

Dabei bedeutet das Zeichen `\n` ein Zeilenvorschubzeichen (*new line*). Es ist hier nötig, da sonst die Ausgaben von aufeinanderfolgenden `printf()`-Aufrufen direkt hintereinander kleben.

Im Gegensatz zu `puts()` erzeugt `printf()` nämlich keinen selbsttätigen Zeilenvorschub.

Die Funktion `printf()` kann aber mehr. Wenn in dem auszugebenden String ein Prozentzeichen `%` eingestreut wird, dann wird an dieser Stelle etwas in die Ausgabe eingefügt. Mit `%d` kann man eine ganze Zahl (also einen Wert vom Datentyp `int`) ausgeben, mit `%f` eine gebrochene Gleitkommazahl, mit `%c` ein Zeichen, mit `%s` einen String und mehr.

Die Werte, die für die Prozentzeichenfolgen eingesetzt werden sollen, folgen beim Aufruf von `printf()` einfach hinter dem auszugebenden String.

Zwischen dem Prozentzeichen und dem Kennbuchstaben, der anzeigt, was denn nun ausgegeben werden soll, kann man noch verschiedene Steuerzeichen einstreuen. Mit `%10.4f` kann man eine gebrochene Zahl (Datentyp `double`) so ausgeben, daß insgesamt 10 Zeichen geschrieben werden (inklusive Vorzeichen, Dezimalpunkt und eventuellen Leerzeichen), davon 4 Nachkommastellen.

Die möglichen Angaben in dem Formatstring werden bei **printf()** (Seite 210) aufgeführt.

Mit dem Programm

```
/* klprfmt.c 30. 7.95 kw
 */

main()
{
  printf( "12345678901234567890\n" );
  printf( "%10.4f\n", 3.14159265354 );
  printf( "%+10.4f\n", 3.14159265354 );
  printf( "%-10.4f\n", 3.14159265354 );
  printf( "%-+10.4f\n", 3.14159265354 );
  printf( "%20.4f\n", 3.14159265354 );
  printf( "%020.4f\n", 3.14159265354 );
}
```

erhält man die Ausgabe:

```
12345678901234567890
    3.1416
   +3.1416
3.1416
+3.1416
              3.1416
000000000000003.1416
```

Der Aufruf

```
printf( "eine ganze Zahl %d und eine halbe %f\n", 2, 0.5 )
```

erzeugt die Ausgabe

```
eine ganze Zahl 2 und eine halbe 0.500000
```

Die Prozentzeichenfolgen im String und die auszugebenden Werte müssen in Reihenfolge und Anzahl zusammenpassen. Der Aufruf

```
printf( "eine ganze Zahl %d und eine halbe %f", 0.5, 2 )
```

würde nur unsinnige Werte ausgeben.

Ein anderes Beispiel für `printf()` ist:

```
/* klq100.c 30. 7.95 kw
 *
 * Dieses Programm gibt die Zahlen von 1 bis 10 und die Quadrate aus.
 */

#define IMAX        10

int printf( char *format, ... );   /* gibt format und eventuell
                                    * weitere Werte auf die Standard-
                                    * ausgabe.
                                    */
```

```
int main()
{
  int     i;                /* eine ganzzahlige Variable als
                             * Schleifenzaehler vereinbaren.
                             */

  printf( "Dieses Programm gibt die ersten %d Quadratzahlen",
          IMAX
          );
  printf( " auf dem Bildschirm aus.\n\n" );

  /* Schleife ueber alle auszugebenden Zahlen:
   */
  for( i=1; i<=IMAX; i=i+1 )
  {
    printf( "Das Quadrat von %d ist %d.\n", i, i*i );
  }

  return 0;                 /* und den Wert 0 zurueckgeben.
                             */
}
```

Die Ausgabe dazu ist:

```
Dieses Programm gibt die ersten 10 Quadratzahlen auf dem Bildschirm aus.

Das Quadrat von 1 ist 1.
Das Quadrat von 2 ist 4.
Das Quadrat von 3 ist 9.
Das Quadrat von 4 ist 16.
Das Quadrat von 5 ist 25.
Das Quadrat von 6 ist 36.
Das Quadrat von 7 ist 49.
Das Quadrat von 8 ist 64.
Das Quadrat von 9 ist 81.
Das Quadrat von 10 ist 100.
```

Obwohl die Obergrenze `10` effektiv an 2 Stellen im Programm verwendet wird, braucht man nur an einer Stelle die `#define`-Anweisung ändern, um statt der ersten 10 die sagen wir ersten 100 Quadratzahlen auf dem Bildschirm sehen zu können.

Wie weit kann man den Parameter `IMAX` erniedrigen?
Wie weit kann man den Parameter `IMAX` erhöhen?

Das hängt vom Wertebereich ab, den der Datentyp `int` auf dem verwendeten System hat. Das betragsmäßig größte Zwischenergebnis tritt im Ausdruck `i*i` auf; hier darf der Wertebereich nicht überschritten werden[2].

Wenn der Wertebereich des Datentyps `int` nicht reicht, um besonders große Zahlen darstellen zu können, kann man stattdessen den Datentyp `long int` oder abgekürzt `long` verwenden. Allerdings muß man dann in der `printf()`-Anweisung statt `%d` die Zeichenfolge `%ld` verwenden.

[2]Das kontrolliert aber kaum ein *C*-Compiler; die Verantwortung dafür liegt wieder einmal beim Programmierer (die löbliche Ausnahme ist der Gnu-C/C++-Compiler `g++`, wenn man ihn mit der Option `-Wall` aufruft).
Informationen zum Wertebereich stehen in **8.4.1 Ganzzahlige Datentypen** auf Seite 39.

Damit hat unser nächstes Beispielprogramm die Form:

```
/* klq1000.c 30. 7.95 kw
 * Dieses Programm gibt die Zahlen von 1 bis 1000 und die Quadrate
 * aus.
 */

#define IMAX        1000

int printf( char *format, ... );   /* gibt format und eventuell
                                    * weitere Werte auf die Standard-
                                    * ausgabe.
                                    */

int main()
{
  long int      i;           /* eine ganzzahlige Variable als
                              * Schleifenzaehler vereinbaren.
                              */

  printf( "Dieses Programm gibt die ersten %d Quadratzahlen",
          IMAX
          );
  printf( " auf dem Bildschirm aus.\n\n" );

  /* Schleife ueber alle auszugebenden Zahlen:
   */
  for( i=1; i<=IMAX; i=i+1 )
  {
    printf( "Das Quadrat von %ld ist %ld.\n", i, i*i );
  }

  return 0;                  /* und den Wert 0 zurueckgeben.    */
}
```

Als nächstes wollen wir nicht nur Quadrate (2. Potenzen), sondern auch die 3. und 4. Potenzen der ersten 10 Zahlen auf dem Bildschirm haben.

Da wir jetzt von jeder Zahl gleich drei verschiedene Potenzen brauchen, lohnt es sich fast, dafür eine eigene Funktion zu schreiben.

Sinnigerweise geben wir ihr den Namen `potenz()`[3].

Das Programm sieht nun so aus:

```
/* klqp4.c 30. 7.95 kw
 *
 * Dieses Programm gibt die Zahlen von 1 bis 10 und die ersten 4
 * Potenzen derselben aus.
 */

#define IMAX        10
```

[3] Ein effektiverer Algorithmus für ganzzahlige Potenzen ist in **A.2 Potenzen mit ganzzahligen Exponenten** auf Seite 245 angegeben.

```
int printf( char *format, ... );   /* gibt format und eventuell
                                    * weitere Werte auf die Standard-
                                    * ausgabe.
                                    */

/* Diese Funktion liefert zu einem Wert die n-te Potenz:
 */
long int potenz( long wert, int n )
{
  long    ergebnis;
  int     i;

  ergebnis = 1;
  for( i=1; i<=n; i=i+1 )
    ergebnis = ergebnis * wert;

  return ergebnis;
}

int main()
{
  int      i;                /* eine ganzzahlige Variable als
                              * Schleifenzaehler vereinbaren.
                              */

  printf( "Dieses Programm gibt die ersten 4 Potenzen zu den" );
  printf( " ersten %d Zahlen aus.\n\n",
          IMAX
          );

  /* Schleife ueber alle auszugebenden Zahlen:
   */
  for( i=1; i<=IMAX; i=i+1 )
  {
    printf( "%d hoch 2 ist %ld, hoch 3 ist %ld, hoch 4 ist %ld\n",
            i,
            potenz( i, 2 ),
            potenz( i, 3 ),
            potenz( i, 4 )
            );
  }

  return 0;                  /* und den Wert 0 zurueckgeben.
                              */
}
```

Definitionen und Deklarationen: Neben den Praeprozessordirektiven und Kommentar finden sich in einem C-Quelltext meist noch *Deklarationen* und *Definitionen* für Daten und Programmcode.

Deklarationen vereinbaren nur das „Aussehen“ von Daten oder Unterprogrammen, während Definitionen auch Platz im Programm dafür schaffen. Dies gilt gleichermaßen für Daten als auch für Unterprogramme (Funktionen).

Der Programmteil

```
int summe( int a, int b );
```

ist nur eine Deklaration. Das heißt für den Compiler, daß es irgendwo eine Funktion namens `summe` gibt, die zwei Argumente vom Typ `int` erhält und einen Wert vom Typ `int` liefert. Wo die Funktion steht, weiß der Compiler damit noch nicht. Die Definition dafür kann später im Quelltext kommen, in einer anderen Datei stehen oder nirgends.

Der Quelltext

```
int summe( int a, int b )
{
    return a + b;
}
```

enthält die gleiche Deklaration wie oben, aber zusätzlich die Definition der Funktion, nämlich: liefere beim Aufruf die Summe der beiden Argumente zurück.

Beachten muß man dabei, daß eine Funktions*deklaration* mit einem Semikolon abgeschlossen wird; eine Funktions*definition* dagegen nicht!

Daten können ähnlich vereinbart werden.

```
int a, b;
```

vereinbart zwei Variablen namens `a` und `b`, beide vom Typ `int`. Der Compiler legt gleich im Programm Platz dafür an und merkt sich die beiden Namen (`a` und `b`) als Synonyme für die beiden Speicherplätze. Dies ist also eine Definition und eine Deklaration gleichzeitig. Mit

```
extern int a, b;
```

dagegen erfolgt nur eine Deklaration. Das heißt, daß der Compiler keinen Speicherplatz reserviert, sondern sich nur die Namen als Variablen vom Typ `int` merkt, aber davon ausgeht, daß die Platzreservierung (also die Definition) anderswo im Programm, eventuell in einer anderen Quelldatei, erfolgt. Der Programmierer ist selbst dafür verantwortlich, daß die zugehörige Definition in der anderen Quelltextdatei identisch ist und nicht möglicherweise mit einem anderen Typ versehen wird. Siehe dazu auch **16 Aufteilung auf mehrere Quelltexte** auf Seite 115.

Anweisungen: Einzelne Anweisungen in einer Funktion werden mit einem Semikolon (`;`) abgeschlossen.[4]

[4] Dies sieht auf den ersten Blick ähnlich aus wie in *Pascal*. In *Pascal* aber ist ein Semikolon nur zum Trennen der Anweisungen nötig, also nicht hinter der letzten Anweisung eines Blocks. In *C* dagegen wird jede vollständige Anweisung mit Semikolon abgeschlossen, also auch die letzte.

4 Quellzeichensatz

C benötigt —wie alle anderen Sprachen auch— zur Darstellung seiner Programme einen bestimmten Zeichensatz.

Dabei wird formal zwischen dem Zeichensatz des eigentlichen Quelltextes *(source character set)* und dem Zeichensatz des Rechners unterschieden, auf dem das übersetzte Programm laufen soll *(target character set)*.

Außer bei sogenannten Crosscompilern ist diese Unterscheidung in aller Regel unwichtig.

Der Quelltext kann aus den 26 Kleinbuchstaben, den 26 Großbuchstaben (beides ohne Umlaute), den 10 Ziffern und folgenden Zeichen bestehen: `!`, `"`, `#`, `%`, `&`, `'`, `(`, `)`, `*`, `+`, `-`, `,`, `\`, `:`, `;`, `<`, `>`, `=`, `?`, `[`, `]`, `/`, `^`, `_`, `{`, `}`, `|`, `~`

Das Leerzeichen, Horizontal- und Vertikaltabulator, sowie Seitenvorschubzeichen *(form feed)* können ebenfalls vorkommen und werden als sogenannte *white space*-Zeichen alle gleich behandelt, nämlich als Trennzeichen zwischen zusammengehörigen Zeichenfolgen (sogenannte *token*). Jeder Name, jeder Operator, jeder String etc. ist dabei ein solches *token*.

Bei den meisten Systemen zusätzlich das Dollarzeichen `$` erlaubt; dies ist aber nicht portabel.

In Kommentaren sowie in Zeichen- und Stringkonstanten werden *white space*-Zeichen in ihrer Form nicht verändert, außerdem dürfen weitere Zeichen des Rechnerzeichensatzes enthalten sein.

Trigraphsequenzen: Normalerweise sollten die Zeichen des *source character set* auf jedem Rechner vorhanden sein, der nicht allzu archaisch ist.

Diese Zeichen sind alle in dem *7-Bit-ASCII*-Zeichensatz enthalten. Eine kaum noch benutzte Untermenge davon, der *ISO 646-1983*-Zeichensatz, enthält nicht alle oben aufgeführten Zeichen. Um auch auf solchen Rechnern[5] die geliebte Sprache *C* wenigstens notdürftig zu ermöglichen, sind die dann fehlenden Zeichen auch anders darstellbar, nämlich mit sogenannten *Trigraphsequenzen*. Diese Zeichenfolgen bestehen aus jeweils zwei Fragezeichen und einem weiteren Zeichen und ersetzen genau ein Zeichen des gewohnten Zeichensatzes:

Trigraphsequenz:	steht für:	Trigraphsequenz:	steht für:
`??(`	`[`	`??=`	`#`
`??)`	`]`	`??/`	`\`
`??<`	`{`	`??!`	`|`
`??>`	`}`	`??'`	`~`

Noch vor jeder weiteren Bearbeitung (also noch vor dem Praeprozessor und dem eigentlichen Compiler) werden im Übersetzungslauf alle Trigraphsequenzen durch das entsprechende Zeichen ersetzt.

[5]Die treibende Kraft der Moderne (mit drei blauen Buchstaben, beginnt mit *I* und endet mit *BM*) hält teilweise noch an derartigem HiTech-Gerät fest.

5 Der Praeprozessor

Vor dem eigentlichen Kompilieren wird ein *C*-Quelltext immer durch den sogenannten *Praeprozessor* geschickt. Der Compiler selbst bekommt den Quelltext nie zu sehen, sondern nur die Ausgabe des Praeprozessors. Den Praeprozessor muß man sich als eigenständiges Programm denken, welches vor dem Compiler aufgerufen wird. In der Praxis sind allerdings beide meistens in einem Programm vereinigt. Prinzipiell ist der Praeprozessor aber nicht an die Sprache *C* gebunden; man könnte mit Hilfe eines eigenständigen Praeprozessors auch die Entwicklung von Programmen in *FORTRAN* oder *Pascal* vereinfachen oder andere Aufgaben der Textmanipulation damit erledigen.

C ist jedoch umgekehrt an den Praeprozessor gebunden, da letzterer sowohl im alten *K&R-C* als auch im *ANSI-C* genormter Bestandteil der Sprache ist.

Die gesamte Tätigkeit des Praeprozessors beschränkt sich auf Textersatz.

Praeprozessoranweisungen beginnen mit einem `#`-Zeichen als erstem Zeichen in der Zeile.

Sofern man nicht mit ausdrücklichen Praeprozessorbefehlen etwas anderes vorgibt, wird der Quelltext unverändert an den Compiler weitergereicht.

Die wichtigsten Praeprozessordirektiven sind:

- `#define`
- `#include <dateiname>`
- `#include "dateiname"`

Mit `#define` kann man zum einen für ein später verwendetes Symbol einen *Namen* vereinbaren und zum anderen sogenannte funktionsähnliche *Makros* definieren. Dann folgt dem Namen des zu vereinbarenden Makros ein Klammerpaar mit einer Liste von Parametern (ohne Typangabe). Zwischen dem Namen und der schließenden Klammer dürfen keine Leerzeichen auftreten. Danach folgt in beiden Varianten die Zeichenfolge, durch die der Makroaufruf in Zukunft ersetzt werden soll.

Die Wirkung von `#define` ist auf den Rest des aktuellen Quelltextes beschränkt, einschließlich der mit `#include` eingebundenen Dateien (siehe unten).

Beispiel:

Aus

```
#define PI   3.1415923564
a = 2.0*PI;
b = 1.0+2*(PI+3);
```

macht der Praeprozessor den Text

```
a = 2.0*3.1415923564;
b = 1.0+2*(3.1415923564+3);
```

Der Quelltext wird dann in dieser Form an den Compiler weitergereicht.

Aus

```
#define sum(a,b)     (a+b)
/* ... */
erg = sum( 1.0, wert );
```

macht der Praeprozessor:

```
/* ... */
erg = (1.0+wert);
```

Man kann auch einen Namen definieren, dem nichts zugeordnet wird.

```
#define unfug
c = unfug - 3.0;
```

wird durch den Praeprozessor zu

```
c = - 3.0;
```

Das macht erst mal keinen Sinn, ist aber für `#ifdef` und `#ifndef` manchmal ganz nützlich; diese werden gleich beschrieben.

Die funktionsähnlichen Makros werden oft eingesetzt, um kurze Funktionen zu ersetzen. Beispielsweise kann

```
int max( int a, int b )
{
    return ( a>b ? a : b );
}
/* ... */
    gross = max(i,gross);
/* ... */
```

ersetzt[6] werden durch:

```
#define max(a,b) ( a>b ? a : b );
/* ... */
gross = max(i,gross);
/* ... */
```

Der Programmtext wird dann vom Praeprozessor ersetzt durch:

```
/* ... */
gross = ( i>gross ? i : gross );
/* ... */
```

[6]Der Operator ... `?` ... `:` ... wertet den Vergleich links des Fragezeichens aus; je nach dem Ergebnis des Vergleichs wird der Wert links (wenn *wahr*) oder der Wert rechts des Doppelpunkts (wenn *falsch*) geliefert.

Diese Form des Programms ist zum einen schneller als die ursprüngliche mit dem Funktionsaufruf, da zur Laufzeit keine Parameter und Rückgabewerte verwaltet werden müssen (sogenannter *inline code*). Zum anderen ist das Makro auch flexibler einsetzbar als die eigentliche Funktion, da man es auch mit anderen Zahlentypen als mit `int` aufrufen kann und entsprechend auch den richtigen Ergebnistyp erhält. `max( 1.0, 2.0 )` liefert ein Ergebnis vom Typ `double`, wenn `max` als Makro vereinbart ist; dagegen würde die oben definierte Funktion `max()` immer ein `int`-Ergebnis liefern.

Funktionsähnliche Makros haben aber ihre Tücken. Die Wirkung im C-Quelltext ist manchmal beileibe nicht dieselbe, wie beim tatsächlichen Aufruf von Funktionen.

Die erste Falle besteht in der Auswertungsreihenfolge von Ausdrücken. Funktionsaufrufe werden immer vor allen Berechnungen ausgeführt.

```
int summe( int a, int b )
{
    return a+b;
}
/* ... */
    c = 5*summe( 1, 3 );
```

berechnet die Summe von `1` und `3`. Danach erst wird das Ergebnis mit `5` multipliziert und der resultierende Wert `20` wird an `c` zugewiesen.

Mit dem Makro

```
#define summe(a,b)   a+b
/* ... */
    c = 5*summe( 1, 3 );
```

glaubt man auf den ersten Blick vielleicht, die gleiche Berechnung durchzuführen. In Wirklichkeit aber wird vom Praeprozessor folgender Text an den Compiler weitergereicht:

```
/* ... */
    c = 5*1+3;
```

Dies ist mitnichten gleichwertig, da durch die Bedingung „Punkt- vor Strichrechnung“ erst `5` mit `1` multipliziert wird, dann wird `3` addiert. Das Ergebnis (`8`) wird an `c` zugewiesen.

Dieses Problem ist aber durch Klammerung leicht zu lösen:

```
#define summe(a,b)   (a+b)
/* ... */
    c = 5*summe( 1, 3 );
```

wird vom Praeprozessor zu

```
/* ... */
    c = 5*(1+3);
```

expandiert und die Welt ist wieder in Ordnung.

Ein ähnlicher Fehler kann auftreten, wenn die Argumente in der Ersatzzeichenfolge nicht geklammert werden:

```
#define produkt(a,b)   (a*b)
/* ... */
    a = 4;
    b = 5;
    c = produkt( a, b+1 );
```

Das wird expandiert zu:

```
/* ... */
    a = 4;
    b = 5;
    c = ( a*b+1 );
```

Durch die Punkt- vor Strichbedingung wird erst `(4*5)` berechnet, dann wird `1` dazugezählt.

Beabsichtigt war wohl eher `4*(5+1)`; dies kann entweder durch Klammerung beim Aufruf:

```
#define produkt(a,b)   (a*b)
/* ... */
    a = 4;
    b = 5;
    c = produkt( a, (b+1) );
```

oder durch Klammerung der Argumente im Makro erreicht werden:

```
#define produkt(a,b)   ((a)*(b))
/* ... */
    a = 4;
    b = 5;
    c = produkt( a, b+1 );
```

Der zweite Weg ist wohl der sicherere.

Funktionsähnliche Makros sollte man also im Zweifelsfall immer fleißig klammern.

Eine weitere Tücke liegt darin, daß Makros einen Parameter eventuell mehrfach auswerten. Wird ein Parameter beim Aufruf manipuliert, beispielsweise durch den Operator `++`, dann wird diese Operation möglicherweise mehrfach durchgeführt.

```
int max( int a, int b )
{
    return ( a>b ? a : b );
}
/* ... */
    gross = max(++i,gross);
/* ... */
```

erhöht `i` um eins und weist an `gross` dann den Funktionswert von `max()` zu.

Mit einem Makro sieht das Programmstück so aus:

```
#define max(a,b) ( a>b ? a : b );
/* ... */
    gross = max(++i,gross);
/* ... */
```

Der Programmtext wird dann vom Praeprozessor ersetzt durch:

```
/* ... */
    gross = ( ++i>gross ? ++i : gross );
/* ... */
```

Dabei wird ebenfalls erst `i` um eins erhöht und mit dem Wert von `gross` verglichen. Ist der Vergleich auf *größer* erfüllt, dann wird `++i` an `gross` zugewiesen, also `i` nochmals erhöht. Ist die Bedingung dagegen nicht erfüllt, dann verhält sich das Makro wie der Funktionsaufruf.

Verhindern kann man den Fehler nur, indem man die Parameter nicht beim Aufruf manipuliert, sondern gegebenenfalls vorher oder nachher:

```
#define max(a,b) ( a>b ? a : b );
/* ... */
    ++i;
    gross = max(i,gross);
/* ... */
```

Reicht eine Zeile nicht aus, um ein Makro zu definieren, dann kann man an das Ende der ersten Zeile das Zeichen `\` stellen und die Definition in der nächsten Zeile fortsetzen; reicht diese auch nicht, dann kann man diese auch wieder fortsetzen etc.; man muß nur aufpassen, daß man hinter die letzte Zeile der Definition kein `\` setzt.

Die Argumente in einem Makroaufruf dürfen übrigens wieder Makroaufrufe enthalten; die enthaltenen Aufrufe werden zuerst ausgewertet und dann erst werden die schon expandierten Argumente in dem äußeren Makro verwendet.

Wenn man Makros mit Argumenten definiert, dann kann man vor jeder Verwendung eines Arguments in der Definition das Zeichen `#` oder die Zeichen `##` vorsetzen. Mit `#` erreicht man, daß das folgende Argument als String expandiert wird: Das Zeichen `#` wird beim Expandieren entfernt und das Argument vor dem Einsetzen mit Anführungsstrichen `"` versehen.

Beispiel:

```
/* pr2str.c 30. 7.95 kw
 */

#define printint(a)   printf( #a " ist %d\n", a )

main()
{
  printint( 2+3 );
}
```

wird vom Praeprozessor zu folgendem expandiert:

```
# 1 "pr2str.c"





main()
{
  printf( "2+3" " ist %d\n",   2+3   ) ;
}
```

Die Ausgabe davon lautet:

```
2+3 ist 5
```

Mit ## kann man ein Argument mit dem vorhergehenden oder dem nachfolgenden Token zu einem verketten.

```
/* prcat.c 30. 7.95 kw
 */

#define var(a) variable ## a

main()
{
  int
    variable1,
    variablexy;

  variable1  = 1;
  variablexy = 25;
  printf( "%d, %d\n", var(1), var(xy) );
}
```

gibt die beiden Variablen `variable1` und `variablexy` aus.

Die (falsche) Version

```
#define var(a) variable a
```

macht aus dem Aufruf

```
printf( "%d, %d\n",  var(1),  var(xy) );
```

den Text

```
printf( "%d, %d\n",  variable 1,  variable xy );
```

und läßt also die einzelnen Token getrennt stehen; mit

```
#define var(a) variable ## a
```

dagegen erhält man aus

```
printf( "%d, %d\n",  var(1),  var(xy) );
```

nach dem Praeprozessor

```
printf( "%d, %d\n",  variable1,  variablexy );
```

und gibt tatsächlich den Wert der beiden Variablen aus.

#undef ist das Gegenstück zu `#define`. Mit `#undef` kann man einen bereits vereinbarten Namen oder ein Makro für den Rest des Quelltextes wieder vergessen lassen; sei es, weil man den Namen nur kurz brauchte oder für einen anderen Zweck neu vergeben will.

Das erneute Vergeben von Namen sollte man aber nur wohlüberlegt einsetzen, da der Quelltext in der Regel unübersichtlich wird, wenn ein und derselbe Name mit verschiedenen Werten und Bedeutungen belegt wird.

Wenn man mit `#undef` eine Definition löscht, die es gar nicht gibt, dann passiert nichts; dies zählt nicht als Fehler. Etwas zu definieren, was bereits definiert ist, ist dagegen ein Fehler. Wenn man also einen Namen definieren möchte, und sich nicht nicht sicher ist, ob er bereits existiert, dann kann man ihn vorher einfach mit #undef löschen, und dann mit #define neu definieren.

#include fügt an der aktuellen Stelle den gesamten Inhalt einer anderen Datei ein. Deren Text durchläuft wiederum den Praeprozessor, bevor er an den Compiler weitergereicht wird. Dadurch kann die eingefügte Datei wieder mit `#include` eine weitere einlesen lassen.

Der Name der einzulesenden Datei muß entweder in ein Paar von spitzen Klammern (`<`...`>`) oder in Gänsefüßchen (`"`...`"`) eingeschlossen werden.

Gibt man beim Dateinamen den vollen Pfadnamen an, dann sind die beiden Formen gleich. Steht vor dem Dateinamen aber kein Pfad, dann wird bei `#include "dateiname"` zuerst da nach der Datei gesucht, wo auch der Quelltext herstammt. Für die Dateien, die mit `#include <dateiname>` eingebunden werden, existiert immer ein vordefinierter Pfad, mit dem diese Dateien gesucht werden.

Bei kommandozeilenorientierten Umgebungen muß man den Pfad meist in einer sogenannten *environment*-Variablen namens `INCLUDE` hinterlegen. Unter MSDOS erreicht man das mit dem Kommando `SET INCLUDE=`*pfad*. Arbeitet man mit Unix, dann muß man meistens ebenfalls mit `set INCLUDE=`*pfad* die Variable setzen, aber eventuell (je nach verwendeter *Shell*) auch noch exportieren mit `export INCLUDE`. In Editor-/Compilerumgebungen wie *Turbo-C* oder *Prospero-C* wird dieser Pfad als Option irgendwo im Menü eingestellt.

Mit `#include <`...`>` fügt man vor allem sogenannte header-Dateien von dritter Seite ein. Das sind Dateien, die beispielsweise vom Compilerhersteller mitgeliefert werden und Vereinbarungen und Makros zur Verwendung der Standardbibliotheken enthalten. Dadurch muß man nicht alle Funktionsdeklarationen und `#define`-Anweisungen selbst schreiben, sondern fügt einfach die entsprechenden Dateien mit `#include` ein.

Mit `#include "`...`"` dagegen werden meist eigene Vereinbarungen eingefügt. Dies ist in der Regel günstiger, als die Vereinbarungen direkt in den Quelltext einzustreuen, da man bei mittleren und größeren Programmen den ganzen Quelltext irgendwann auf mehrere Dateien verteilt. Ändert sich irgendeine Vereinbarung dann, müßte man diese in allen Quelltexten korrigieren. Steht anstatt der Vereinbarungen in jedem Quelltext dagegen nur `#include "`*sowieso*`"`, dann kommen alle Vereinbarungen nur einmal vor und brauchen auch nur an einer Stelle geändert werden.

Dadurch spart man letztlich viel Arbeit und hat wesentlich weniger Fehlerquellen. Siehe dazu die Ausführungen in **16 Aufteilung auf mehrere Quelltexte** auf Seite 115.

#if und seine Anverwandten (`#elif`, `#else`, `#endif`, `#ifdef`, `#ifndef`) kann man für bedingte Kompilierung nutzen.

Als Bedingung darf ein *C*-ähnlicher logischer Ausdruck verwendet werden, der allerdings nur aus Konstanten, Praeprozessormakros und den Operatoren bestehen darf.

Außerdem kann man in der `#if`-Bedingung mit `defined` feststellen, ob ein Name als Makro vereinbart ist.

Bei

```
#define MEIER
#if defined MEIER
    tuwas
#endif
```

wird `tuwas` an den Compiler gereicht, bei

```
#if defined MEIER
    tuwas
#endif
```

(ohne vorheriges `#define MEIER`) dagegen nicht.

`#if defined ...` kann zu `#ifdef ...` abgekürzt werden, hier also bei dem obigen Beispiel:

```
#define MEIER
#ifdef MEIER
    tuwas
#endif
```

Analog wird der Teil, der hinter `#ifndef ...` steht, genau dann ausgeführt, wenn der angegebene Name *nicht* definiert ist.

Der Praeprozessor wird oft dazu benutzt, um verschiedene Versionen eines Programms in einem Quelltext halten zu können und mit Änderung einer `#define`-Anweisung von einer Version zu einer anderen wechseln zu können. Beispielsweise kann man Unterschiede zwischen verschiedenen Rechnersystemen etwa so berücksichtigen:

```
#define TURBOC     1
#define PROSPEROC  2
#define VMSC       3
#define WELCHESC   TURBOC
/* ... */
#if WELCHESC==TURBOC
     teil1
#elif WELCHESC==PROSPEROC
     teil2
#elif WELCHESC==VMSC
     teil3
```

```
#else
     teil4
#endif
/* ... */
```

Da hier `WELCHESC` als `TURBOC` definiert ist, wird nur `teil1` an den Compiler weitergereicht. Die Programmstücke `teil2`, `teil3` und `teil4` werden dem Compiler vom Praeprozessor vorenthalten und folglich nicht mitübersetzt. So kann man mit Hilfe von Praeprozessordirektiven in einem Quelltext verschiedene Versionen eines Programms halten (beispielsweise für verschiedene Rechnersysteme oder verschiedene Kunden) und die jeweilige Version nur an einer Stelle mit einem `#define` auswählen.

Oft wird ein Programm während der Entwicklung mit ausführlicheren Fehlermeldungen versehen, die mit

```
#ifdef DEBUG
    /* ... */
#endif
```

geklammert sind. Will man die Fehlermeldungen für Testzwecke haben, schreibt man an den Anfang des Quelltextes `#define DEBUG` und hat nach dem Übersetzen die entsprechenden Programmteile mitübersetzt. Ist das Programm fertig ausgetestet, dann kann man `#define DEBUG` wieder entfernen und hat nach dem Übersetzen das entsprechend kleinere Programm zur Verfügung. Treten Fehler auf: kein Problem, `#define DEBUG` wieder einsetzen und schon kann man weiter austesten!

Es hat sich als sehr zweckmässig erwiesen, während der Programmentwicklung die Anweisung `#define DEBUG` an Beginn eines jeden Quelltextes beziehungsweise einer überall eingebundenen Headerdatei zu stellen und nach Entwicklungsende für die Produktionsversion die Zeile gar nicht erst zu entfernen, sondern in `#define NDEBUG` umzuändern. Zum einen kann man so zwischen Test- und Produktionsversion jeweils mit einem Tastendruck wechseln, zum anderen hat man dadurch in der Produktionsversion gleichzeitig alle `assert()`-Aufrufe stillgelegt (siehe **`assert()`** (Seite 178) beziehungsweise **17 Fehlersuche mit Standard-*C*** auf Seite 130).

#line manipuliert die Zeilennumerierung des Compilers.

Angenommen, in der Zeile *25* steht die Direktive

```
#line 2000
```

, dann zählt der Compiler ab dieser Stelle die Zeilennummern weiter, als ob die Zeile mit der `#line`-Anweisung die Nummer *2000* gehabt hätte. Die folgende Zeile wird *2001*, und so fort.

#error gibt während der Übersetzung eine Fehlermeldung auf die Standardfehlerausgabe (inklusive der Token, die der `#error`-Anweisung folgen) und beendet die Kompilation.

Beispiel:

```
#error Mist, alles Muell!
```

gibt den Text `Mist, alles Muell` aus und beendet die Übersetzung.

Diese Anweisung ist beispielsweise sinnvoll, wenn man mehrere Versionen eines Programms in einem Quelltext hält und mit Praeprozessordirektiven dazwischen umschalten kann. Dann kann es vorkommen, daß man bei Erweiterungen der Variationen an einer Stelle die neue Variante vergißt. Dies kann man von vornherein unterbinden, wenn man in alle `#if` – `#elif` – `#endif`-Entscheidungen noch einen `#else`-Zweig mit einer `#error`-Anweisung einsetzt.

Beispiel:

```
#define     ATARI   1
#define     MSDOOF  2
#define     BSD_V   3

#define     RECHNER     ATARI   /* ATARI, MSDOOF, MSDOOF */

/* ... */

#if RECHNER==ATARI
    /* Atari-spezifisches
     */
    /* ... */
#elif RECHNER==MSDOOF
    /* Pech gehabt!
     */
    /* ... */
#elif RECHNER==BSD_V
    /* Unix-spezifisches
     */
    /* ... */
#else
    #error RECHNER ist kein gueltiger Rechner!
#endif
/* ... */
```

Wird dann zu einem späteren Zeitpunkt ein weiterer Rechnertyp angefügt (zum Beispiel mit `#define SUN 4`) und mit `#define RECHNER SUN` auch ausprobiert, dann vergißt man natürlich leicht, eine der `#if`-Entscheidungen zu ergänzen. Dann kommt die `#error`-Anweisung zum Zuge und die Kompilation wird mit Angabe der Fehlerstelle und dem Text:

```
4 ist kein gueltiger Rechner!
```

abgebrochen.

#pragma ist vorgesehen, um bestimmten Compilern Mitteilungen zu machen. Ein Compiler darf jede `#pragma`-Direktive übergehen, wenn er will, oder ihren Inhalt nicht versteht. Die Verwendung ist also systemabhängig und hier nicht von Interesse.

Eine mögliche Anwendung ist, einen bestimmten Compiler irgendwo im Quelltext zu bestimmten Optimierungen zu ermuntern oder davon abzuhalten.

Vordefinierte Makros sind `__LINE__`, `__FILE__`, `__DATE__` und `__TIME__`. Sie sind während der Übersetzung schon vordefiniert und enthalten die aktuelle Zeilennummer (als ganze Zahl), den Namen der gerade übersetzten Datei (als Stringkonstante), das aktuelle Datum (als Stringkonstante, z.B. `"Jan 01 1991"`) bzw. die aktuelle Uhrzeit (ebenfalls als Stringkonstante).

`__STDC__` ist mit `1` definiert, wenn der verwendete Compiler den ANSI-Standard unterstützt, ansonsten nicht.

Diese Makros dürfen nur verwendet werden, aber nicht manipuliert!

Je nach Compilersystem können noch weitere Namen definiert sein.

6 Kommentare

In allen Sprachen ist natürlich eine ausführliche *Kommentierung* dessen, was im Quelltext so alles passiert, sehr wichtig. In *C* beginnen Kommentare mit der Zeichenfolge /* und enden entsprechend mit */.

Alles, was zwischen /* und */ steht, wird vom Compiler also einfach ignoriert und ist nur für dem mehr oder weniger menschlichen Leser des Quelltextes gedacht. Kommentare dürfen sich dabei auch über mehrere Programmzeilen erstrecken.

Bei mehrzeiligen Kommentaren ist es sinnvoll, den Kommentarblock als solchen optisch hervorzuheben, anstatt nur nur den Text in /* und */ einzufassen.

Mehrzeilige Kommentare werden deshalb von manchen Programmierern etwa so geschrieben:

```
/* Das ist ein Kommentar, der -warum auch immer- ueber  */
/* mehrere Zeilen geht.                                 */
/* Das muss aber nicht immer besser sein, als ein       */
/* kurzer Kommentar: wer liest schon so viel Text, ohne */
/* muede zu werden?                                     */
```

Das ist aber etwas langwierig zu tippen; besonders bei kleinen Textänderungen schreibt man sich die Finger wund. Deshalb findet man meistens diese Form:

```
/* Das ist ein Kommentar, der -warum auch immer- ueber
 * mehrere Zeilen geht.
 * Das muss aber nicht immer besser sein, als ein
 * kurzer Kommentar: wer liest schon so viel Text, ohne
 * muede zu werden?
 */
```

Bei manchen C-Compilern ist es erlaubt, innerhalb von Kommentaren wiederum ein Paar von /* und */ stehen zu haben (geschachtelter Kommentar), also eine Konstruktion wie:

```
   /* aeusserer Kommentar /* innerer Kommentar */ aeusserer Kommentar */
```

.

Andere Compiler sehen das zweite /* nicht, weil es ja im Kommentar steht und sehen beim ersten */, den Kommentar als beendet an. Damit steht der Text jkl */ schon außerhalb des Kommentars. Deshalb sollte man Kommentare nie schachteln.

Schachteln von Kommentaren könnte man —sofern der Compiler es zuläßt— nutzen, um „schnell mal" ein Programmstück auszukommentieren (in dem wiederum Kommentare enthalten sind), ohne den Quelltext dauerhaft zu löschen. In diesem Fall kann man aber mit bedingter Kompilierung (siehe Seite **5 Der Praeprozessor** auf Seite 24) wesentlich flexibler und systemunabhängig arbeiten.

Um ein Programmstück stillzulegen, bietet sich folgende Sequenz an:

```
#if 0
    /* stillgelegter Teil */
    ...
#endif
```

Der Wert `0` ist in *C* als *logisch falsch* definiert; siehe dazu **8.4.2 Logischer Datentyp** auf Seite 39. Auch der Praeprozessor hält sich an diese Konvention.

Hinweis: In *C++* gibt es eine weitere Form von Kommentaren. Dabei wird jeglicher Text, der nach zwei direkt aufeinander folgenden Schrägstrichen steht (`//`), bis zum Ende der aktuellen Zeile als Kommentar betrachtet, also ignoriert.

Da es kaum noch *C*-Compiler gibt, die nicht auch *C++* verstehen, sind auf den meisten Systemen auch in *C*-Programmen beide Formen von Kommentar erlaubt. Dies entspricht dann allerdings nicht mehr dem ANSI-Standard.

7 Namen

Namen für Variablen, Funktionen etc. können in *C* recht lang sein. Wie lang, hängt vom jeweiligen System ab. *ANSI-C* fordert, daß Namen mindestens 32 Zeichen lang sein dürfen. Sie dürfen die Buchstaben `a`–`z`, `A`–`Z`, die Ziffern `0`–`9` und den Unterstrich `_` enthalten[7], aber nicht mit einer Ziffer beginnen; ein Unterstrich (`_`) zu Beginn ist nicht portabel, weil es gelegentlich mit systemintern verwendeten Namen kollidiert.

Im Gegensatz zu *Pascal* und *FORTRAN* wird zwischen Groß- und Kleinschreibung unterschieden. `iii` ist also ein anderer Name als `III` oder `iIi`.

Dabei muß man aber beachten, daß in *C* zwischen internen und externen Namen unterschieden wird. Interne Namen sind solche, die nur innerhalb eines Quelltextes verwendet werden. Für diese gelten keine weiteren Einschränkungen.

Externe Namen dagegen sind solche, die der Linker des jeweiligen Rechnersystems bearbeiten muß. Bei solchen kann man sich nicht darauf verlassen, daß 32 signifikante Zeichen zulässig sind. Diese Namen sollten sich in den ersten 6 Zeichen unterscheiden, um Verwechslungen des Linkers zu vermeiden. Außerdem kann es sein, daß der Linker nicht zwischen Groß- und Kleinbuchstaben unterscheidet.

Die Schlüsselworte von *C* dürfen als Namen nicht verwendet werden:

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

Diese Namen sind reservierte Schlüsselworte von *C* und werden nicht als benutzerdefinierte Namen erkannt.

Ebenfalls vermeiden sollte man die Namen

`asm`	`fortran`	`pascal`

sowie alle Namen, die mit einem Tiefstrich `_` beginnen. Diese Namen sind nach *ANSI-C* zwar problemlos, aber viele Implementationen kommen ohne sie nicht aus.

[7] In alten (K&R-*C*-Compilern war auch `$`) in Namen erlaubt (auch am Namensanfang); ANSI-*C* erlaubt dies nicht. Auch wenn die meisten Compiler es dann doch zulassen (zumindest mit Anwendung von Gewalt in Form einer Compileroption), sollte man es soweit möglich vermeiden.

8 Daten

Innerhalb eines Programms werden Daten im Arbeitsspeicher des verwendeten Rechners gehalten.

Dazu können **Variablen** (Seite 35) vereinbart werden, so daß man die Speicherstellen, wo die Daten zu finden sind, mit einem selbstdefinierten Namen ansprechen kann.

Andere Möglichkeiten der Datenspeicherung im Programm sind **Konstanten** (Seite 31), und *allokierter* **Freier Speicher** (Seite 48).

8.1 Konstanten

Literale Konstanten besitzen einen Typ und einen Inhalt. Der Typ ergibt sich aus der Schreibweise, wie im Folgenden beschrieben ist.

Zudem gibt es *benannte Konstanten*, die sich zusätzlich durch einen Namen auszeichnen. Sie werden definiert wie **Variablen** (Seite 35), aber zusätzlich mit dem Schlüsselwort `const` deklariert und sind damit nach der Vereinbarung nicht mehr änderbar. Siehe dazu auch **8.6 Attributangaben** auf Seite 50.

Den Wert von Konstanten darf man durch Zuweisungen o. ä. nicht ändern[8].

Mit der Ausnahme von *Stringkonstanten* haben alle Konstanten einen einfachen Datentyp. Stringkonstanten haben einen zusammengesetzten Datentyp, nämlich ein *Feld vom einfachen Typ* `char`, also den Typ `char[]`.

Ganzzahlige Konstanten schreibt man meist als *Dezimalzahl*. `123` ist beispielsweise eine solche Konstante.

Vorwiegend bei systemnaher Programmierung tut man sich mit *oktaler* oder *sedezimaler* (oder: hexadezimaler) Schreibweise (zur Zahlenbasis 8 bzw. 16) manchmal leichter. Oktale Konstanten beginnt man mit der Ziffer `0`, also etwa `0312`. Das ist oktal und entspricht der Dezimalzahl $(3*8*8+1*8+2) = 202$ (dezimal). Oktalkonstanten dürfen natürlich nur die Ziffern `0`–`7` beinhalten (zumindest in *ANSI-C*; in *K&R-C* waren `8` und `9` als Oktalziffern teilweise erlaubt).

Sedezimalzahlen beginnt man mit einer der Zeichenfolgen `0x` oder `0X`. Als Ziffern braucht man dazu 16 verschiedene Zeichen. Dafür nimmt man `0`–`9` und `a`–`f` oder wahlweise `A`–`F`. `A` und `a` entsprechen dabei dem Wert 10, `B` und `b` dem Wert 11, `C` und `c` dem Wert 12, `D` und `d` dem Wert 13, `E` und `e` dem Wert 14 und `F` und `f` dem Wert 15. Beispielsweise ist `0x75af` eine Sedezimalkonstante und entspricht der Dezimalzahl $(7*16*16*16+5*16*16+10*16+15) = 30127$.

Binäre Zahlen kann man leider nicht im Quelltext angeben.

In der Regel wird eine dezimal geschriebene ganzzahlige Konstante als Konstante mit dem Typ `signed int` angesehen. Ist sie jedoch so groß, daß sie über den Wertebereich von `signed int` hinausgeht, dann wird sie automatisch als `unsigned int` (aber nur, wenn sie sedezimal oder oktal angegeben ist), oder `long int` oder `unsigned long int` behandelt.

Bei kleineren Zahlen kann man das Attribut `unsigned` erzwingen, indem man an die Konstante den Buchstaben `u` oder `U` anhängt. Ebenso kann man durch Anhängen von `l` oder `L` den Typ `long` erzwingen, mit `ul` entsprechend `unsigned long`.
`12ul` ist also eine `unsigned long`-Konstante mit dem Wert `12`.

[8] In Einzelfällen kann man es aber: Man schreibt eine Stringkonstante und läßt einen Zeiger darauf zeigen. Über den Zeiger kann man dann bei einigen Systemen ungestraft den Inhalt der Stringkonstanten ändern; andere Systeme reagieren darauf mit einem Fehler zur Laufzeit.

Aufzählungen: Man kann an Konstanten vom Typ `int` Namen vergeben, indem man sie in einer `enum`-Liste aufführt. Beispiel:

```
enum farbe
{
    rot,
    gruen,
    blau
};

enum farbe col;

main()
{
    col = blau;

    printf( "col ist %d\n", col );
}
```

vereinbart drei `int`-Konstanten, nämlich `rot` mit dem Wert `0`, `gruen` mit dem Wert 1 und `blau` mit dem Wert `2`. Gleichzeitig wird der Datentyp `enum farbe` vereinbart, der damit als `int` zu verstehen ist. Mit diesem Datentyp wird in der folgenden Zeile eine Variable `col` vereinbart. Dieser wird in `main()` der Wert `2` zugewiesen (`blau`).

In obigem Beispiel ist wahrscheinlich beabsichtigt, daß die Variable `col` nur die Werte `0`, `1` und `2` beziehungsweise `rot`, `gruen` und `blau` annehmen kann. Tatsächlich ist `col` vom Typ `int` und kann damit auch alle anderen Werte annehmen, ohne daß der Compiler dies verhindern würde. Die Zuweisung `col=25;` wäre also zulässig, aber wohl unsinnig.

Will man keinen Datentyp `enum soundso` schaffen, dann kann man auch den Namen (hier: `farbe`) weglassen:

```
enum
{
    rot,
    gruen,
    blau
};

int col;

main()
{
    col = blau;

    printf( "col ist %d\n", col );
}
```

Wenn nichts anderes angegeben wird, dann erhält die erste Konstante in der `enum`-Liste den Wert `0` und alle anderen Konstanten erhalten den um eins erhöhten Wert des Vorgängers. Diese Vorgabe kann man aber übersteuern, indem man einige oder alle Werte angibt:

```
enum
{
    rot=3,
    gruen,
    blau=10
};
```

Hier wird `rot` als `3` vereinbart, `gruen` als `4` (eins mehr als der Vorgänger `rot`) und `blau` als `10`.

Konstanten vom Typ `char` kann man schreiben, indem man das entsprechende Zeichen in Apostrophen einschließt. Die Konstante `'A'` ist dafür ein Beispiel. Eine `char`-Konstante hat einen Zahlenwert, welcher der Position des Zeichens in der verwendeten Zeichensatztabelle entspricht. Bei den meisten Rechnern ist dies der *ASCII-Zeichensatz*, wo ein `'A'` den Wert `65` hat. Etliche Zeichen kann man nicht über die Tastatur in den Quelltext eingeben; dafür kann man dann Sonderdarstellungen verwenden:

Zeichen	*Schreibweise*
Zeilenvorschub, Zeilentrenner	`'\n'`
Alarmzeichen	`'\a'`
Tabulatorzeichen	`'\t'`
Vertikaltabulator	`'\v'`
backspace	`'\b'`
carriage return	`'\r'`
Seitenvorschub (*form feed*)	`'\f'`
linker Schrägstrich (*back slash*)	`'\\'`
Fragezeichen	`'\?'`
Apostroph	`'\''`
Gänsefüßchen	`'\"'`

Jedes beliebige Zeichen kann man auch mit seiner oktalen oder sedezimalen Position im Zeichensatz angeben. Oktal gibt man dafür nach dem linken Schrägstrich maximal 3 oktale Ziffern (`0`–`7`) ein, wie: `'\40'` ist im ASCII-Zeichensatz das Leerzeichen `' '`, da es an der Position 32 (dezimal) beziehungsweise 20 (sedezimal) oder 40 (oktal) steht. Die sedezimale Darstellung beginnt mit `\x` und enthält danach maximal 2 sedezimale Ziffern, z.B. `'\x20'`.

Achtung! Diese Positionen sind aber abhängig vom verwendeten Rechnersystem, während die Darstellungen `'\n'` bis `'\"'` in obiger Tabelle rechnerunabhängig sind.

Stringkonstanten sind Konstanten, die den Typ `char[]` haben, also ein Feld von `char`-Werten. Die benötigte Länge des Feldes ist immer (*Anzahl der angegebenen Zeichen* +1). Die um 1 größere Länge kommt daher, daß an den eigentlichen String noch ein Nullbyte angehängt wird, also das Zeichen `'\0'`. Damit wird das Ende des Strings markiert. Beispiel: `"abc"` ist eine Stringkonstante mit 4 Elementen vom Typ `char`, nämlich `'a'`, `'b'`, `'c'` und `'\0'`. In den String dürfen die Steuerzeichen und Sonderzeichen, die bei `char`-Konstanten beschrieben wurden, eingestreut werden: `"a\\b\n"` besteht aus 5 Zeichen, nämlich `'a'`, dem linken Schrägstrich `'\\'`, `'b'`, einem Zeilenvorschubzeichen `'\n'` und der abschließenden `'\0'`.

Achtung: Die Zeichen `'0'` und `'\0'` haben nichts miteinander zu tun!

`'0'` ist das Zeichen im jeweiligen Zeichensatz, welches bei der Ausgabe die Ziffer 0 repräsentiert. Im ASCII-Zeichensatz ist dies das Zeichen mit der Nummer 48.

`'\0'` dagegen ist das Zeichen an der ersten Position, also die Position 0. Bei den meisten Zeichensätzen kann `'\0'` gar nicht dargestellt werden. Sonst könnte man es in *C* auch nicht als Stringende verwenden, denn wie sollte man das Zeichen dann in einem String auf den Bildschirm bringen, wenn es doch das Ende der Zeichenkette signalisiert?

Nach *ANSI-C*[9] kann man mehrere Stringkonstanten hintereinanderschreiben; der Compiler faßt diese dann zu einer zusammen und hängt erst hinter die letzte die abschließende Null. Dies ist ganz nützlich, um in einem Quelltext einen String mit Praeprozessordirektiven zusammenzusetzen.

Beispiel:

```
#define  DATEINAME      "TEST.DAT"
    /* ... */
    FILE *f;

    f = fopen( DATEINAME, "r" ); /* Datei oeffnen                   */

    if( !f )                     /* Oeffnen hat nicht geklappt      */
    {
        puts( "Die Datei " DATEINAME " kann ich nicht oeffnen!" );
    }
    /* ... */
```

An der Stelle, an der `puts()` aufgerufen wird, wird vom Praeprozessor der Text `DATEINAME` ersetzt durch ¨`TEST.DAT`¨. Die Anweisung

```
puts( "Die Datei " "TEST.DAT" " kann ich nicht oeffnen!" );
```

wird übersetzt wie

```
puts( "Die Datei TEST.DAT kann ich nicht oeffnen!" );
```

da die drei Strings im Argument zu einem zusammengefaßt werden.

Wenn man in die Parameterliste einer Funktion eine Stringkonstante oder ein anderes Feld schreibt, dann wird tatsächlich nicht das Feld an die Funktion übergeben, sondern stattdessen ein **Zeiger** (Seite 44) auf das erste Element des Feldes.

Vorsicht! `'A'` und `"A"` sind grundverschiedene Konstanten! `'A'` ist eine Zeichenkonstante, also ein einzelnes Zeichen. `"A"` dagegen ist ein Feld vom Typ `char` („nullterminierter String") mit 2 Zeichen; und zwar dem Zeichen `'A'` und einem anschließenden Nullbyte.

Konstanten vom Typ `double` (doppelt genaue Gleitkommazahlen) schreibt man als Ziffernfolge mit Dezimalpunkt und/oder Zehnerexponent. Beispiele:

- `1.0`
- `1.`
- `0.5`
- `.5`

---

[9] Achtung! Nicht bei älteren *C*-Compilern!

- `3.14159265`
- `0.314159265e1`
- `2.E5`
- `2E5`

und so fort. Mit dem Zusatz `f` oder `F` erzwingt man eine Konstante einfacher Genauigkeit (`float`, z. B. `1.23456F`); mit dem Zusatz `l` oder `L` eine solche mit dem Typ `long double`.

8.2 Variablen

Variablen bestehen aus einem Namen, sowie einem Typ und haben zur Laufzeit des Programms auch noch einen Inhalt (Wert). Weiterhin liegen sie während des Programmlaufs an einer bestimmten Adresse im Speicher (Ausnahme: `register`-Variablen), so daß man auch diese Adresse bestimmen kann.

Variablen müssen in *C* —im Gegensatz zu *FORTRAN*— immer vereinbart werden.

Der Typ einer Variablen ist meist einer der *einfachen Datentypen* oder ein *zusammengesetzter Datentyp*.

Zusammengesetzte Datentypen sind Felder, die aus lauter gleichen Elementen bestehen, oder Strukturen, die auch aus unterschiedlichen Elementen bestehen können. Eine Abart der Strukturen sind die `union`s, die ebenfalls aus mehreren Elementen bestehen; die einzelnen Elemente belegen aber alle ein und dieselbe Adresse im Speicher, so daß immer nur eines der Elemente zu einer Zeit gespeichert werden kann.

Bei Feldern spricht man die einzelnen Elemente über einen Index an, welcher die gewünschte Position im Feld angibt. Dies kann irgendein Ausdruck sein, der zu einem ganzzahligen Wert konvertiert werden kann, also eine Konstante, eine Variable oder auch ein Rechenausdruck. Der Index steht hinter dem Namen der Feldvariablen in eckigen Klammern. Näheres zu Feldern steht in **8.8.1 Felder** auf Seite 61.

Die Elemente einer Struktur oder einer `union` kann man dagegen nur über den Namen des Elements ansprechen, der an den jeweiligen Namen der Variablen mit einem Punkt angehängt wird; siehe dazu **8.7 struct und union** auf Seite 53.

8.3 Gültigkeit, Blöcke

Alle Vereinbarungen (außer Praeprozessoranweisungen) gelten ab ihrer Vereinbarung in dem kleinsten umgebenden Block. Ein Block ist ein Paar von geschweiften Klammern ({...}), in welchem Vereinbarungen (Deklarationen, Definitionen) und Anweisungen stehen dürfen. Eventuelle Vereinbarungen müssen vor den ausführbaren Anweisungen eines Blocks stehen.

Steht eine Vereinbarung außerhalb aller Blöcke, dann gilt sie bis zum Ende der Quelldatei.

Auch jede Variable hat einen Gültigkeitsbereich. Zum einen gilt eine Variable erst ab der Stelle im Quelltext, an der sie vereinbart ist. Zum anderen ist die Wirkung ihrer Vereinbarung auf den kleinsten umschließenden Block beschränkt. Variablen, die außerhalb einer Funktion vereinbart werden, gelten im gesamten weiteren Quelltext (globale Variable).

Wird eine Variable innerhalb einer Funktion vereinbart, dann gilt sie auch nur in dieser Funktion (lokale Variable).

Wird eine Variable innerhalb eines Blockes innerhalb einer Funktion vereinbart, dann gilt sie entsprechend nur in diesem Block.

Vereinbarungen in einem Block müssen vor den ausführbaren Anweisungen des Blocks stehen.

Innerhalb des gleichen Geltungsbereichs darf ein Name nur für eine Variable oder eine Funktion oder einen Datentyp verwendet werden. Jedoch darf in einem Block eine Variable mit einem Namen vereinbart werden, der schon außerhalb des Block vergeben ist. Dann bezieht sich der Name innerhalb des Blocks nur auf die lokale Variable des Blocks.

Die `enum`, `struct`- und `union`-Namen sowie ihre Elementnamen kollidieren nicht mit Variablen- und Funktionsnamen.

Beispiel: Nach den Vereinbarungen

```
enum ename
{
    rot,
    gruen,
    blau
}
farbe;

struct sname
{
    int     i;
    double  d;
}
a, b;
```

dürfen die Namen `ename`, `sname`, `i` und `d` für Variablen im gleichen Geltungsbereich verwendet werden. `rot`, `gruen`, `blau` (an Konstanten vergeben), `farbe`, `a` und `b` (bereits Variablen) dürfen nicht an weitere Variablen oder Konstanten vergeben werden.

Wie lokale Variablen sind die Formalparameter einer Funktion auf die zugehörige Funktion beschränkt, und können ansonsten wie Variablen behandelt werden.

Beispiel:

```
int summe( int a, int b )
{
  return a + b;
}

int differenz( int a, int b )
{
  int minusb;

  minusb = -b;
  return summe( a, minusb );
}

int wert;

rechne()
{
  wert = summe( 1, 2 );
}

main()
{
  int wert;
  rechne();
  wert = differenz( 1, 2 );
}
```

Hier haben die Funktionen `summe()` und `differenz()` je zwei Parameter mit den Namen `a` und `b`, die jedoch außer einer zufällig gleichen Vereinbarung nichts miteinander zu tun haben. Die Variablen `a` und `b` der Funktion `summe()` sind nicht die selben wie die der Funktion `differenz()`! Als Funktionsparameter sind sie lokal auf die zugehörige Funktion beschränkt.

`minusb` ist eine lokale Variable der Funktion `differenz()` und nur innerhalb des geschweiften Klammerpaares {...} von `differenz()` gültig.

`wert` ist zum einen eine globale Variable, die ab ihrer Vereinbarung für alle nachfolgenden Funktionen gelten kann. `rechne()` verwendet sie auch. `main()` dagegen übersteuert die globale Vereinbarung von `wert` mit einer lokalen Vereinbarung und kann damit nicht mehr auf die globale Variable `wert` zugreifen. Die Zuweisung an `wert` innerhalb von `main()` bezieht sich auf die lokale Variable.

8.4 Grunddatentypen

Die Datentypen in *C* sind zum einen die elementaren Grunddatentypen (`signed char` und `unsigned char`, `signed`/`unsigned int`, `signed`/`unsigned long int`, `float`, `double`, `long double`, *Zeiger*, `size_t`, `ptrdiff_t`), sowie daraus zusammengesetzte Datentypen (**8.8.1 Felder** auf Seite 61, **8.7 struct und union** auf Seite 53). Mit `void` als Datentyp kann man anzeigen, daß eine Funktion keinen Wert liefert[10]. Eine *C*-Funktion mit `void` als Ergebnistyp entspricht also einer `PROCEDURE` in *Pascal* oder einer `SUBROUTINE` in *FORTRAN*.

8.4.1 Ganzzahlige Datentypen:

Der wichtigste ganzzahlige Datentyp ist `int`. Damit kann man die üblicherweise auftretenden ganzen Zahlen beschreiben. Von dem Datentyp `int` gibt es verschiedene Abarten, die auch kombiniert werden können, soweit sie sich nicht widersprechen. `short int` ist die platzsparende Version von `int`, `long int` eine Version, die meist mehr Platz braucht, aber dafür einen größeren Zahlenbereich abdecken kann. `long int` ist auf den gängigen Rechnern 4 Byte lang und kann damit den Bereich von `-2147483648` bis `+2147483647` beschreiben.

`int` ist die natürliche Größe des verwendeten Prozessors und damit in der Regel 2 oder 4 Byte lang. Zwei Byte lange `int` können Werte von `-32768` bis `+32767` annehmen. `short int` ist meist 1 oder 2 Byte lang.

Der kleinste ganzzahlige Typ ist `char`. `char` ist nicht auf darstellbare Zeichen beschränkt wie bei Pascal, sondern ein normaler ganzzahliger Datentyp, mit dem man auch rechnen kann.

Beispielsweise ergibt der Ausdruck `('B'-'A')` den Abstand zwischen `B` und `A` im verwendeten Zeichensatz (in aller Regel ist das der Wert 1).

`short int` muß nicht kürzer sein als `int`, `int` muß nicht kürzer sein als `long`. Der ANSI-Standard läßt die genauen Längen offen und schreibt nur vor, daß die Größe von `char` kleiner oder gleich als die von `short int` ist, diese wiederum kleiner oder gleich als die von `int` und diese kleiner oder gleich als die einer `long int`.

Eine weitere Differenzierung hat man mit dem Zusatz `signed` oder `unsigned` zur Verfügung.

Ob der Datentyp `char` eine vorzeichenbehaftete Zahl ist oder vorzeichenlos, legt der ANSI-Standard nicht fest. Bei einem Byte für eine `char`-Zahl kann es also sein, daß man auf einem Rechner für `char` einen Wertebereich von `-128` bis `+127` hat (`signed`), auf einem anderen dagegen den Bereich `0`...`255` (`unsigned`). Mit `signed char` bzw. `unsigned char` kann man den einen oder den anderen Wertebereich erzwingen. Verwendet man als Datentyp `char` ohne den Zusatz `signed` oder `unsigned`, dann kann man sich nur darauf verlasssen, daß man den Bereich von `0` bis `+127` zur Verfügung hat.

Alle anderen ganzzahligen Datentypen sind `signed`, sofern man nicht ausdrücklich `unsigned` angibt. Es ist also gleichgültig, ob ich eine Variable so: `int i;` oder so: `signed int i;` vereinbare; in beiden Fällen ist sie vorzeichenbehaftet.

[10] z.B. ist **qsort()** (Seite 213) aus der Standardbibliothek als `void qsort(...)` deklariert.

Bei jeder ANSI-konformen Implementation sind in der Datei limits.h die Wertebereiche aller ganzzahligen Datentypen eingetragen. Bei 2 Byte langen`int` findet man dort beispielsweise die Zeilen:

```
#define INT_MIN      -32768   /* Achtung! Nur Beispielwerte! */
#define INT_MAX      +32767
```

und für die anderen ganzzahligen Typen analog.

`CHAR_BIT` gibt die Anzahl der Bits in einem `char` an, also meist 8.

`CHAR_MIN`, `CHAR_MAX`, `UCHAR_MAX`, `SCHAR_MIN`, `SCHAR_MAX` geben die Wertebereiche für die Datentypen `char`/`unsigned char`/`signed char` an.

`SHRT_MIN`, `SHRT_MAX`, `USHRT_MAX`, `INT_MIN`, `INT_MAX`, `UINT_MAX`, `LONG_MIN`, `LONG_MAX` und schließlich `ULONG_MAX` enthalten die Wertebereiche für `signed`/`unsigned short` beziehungsweise für `signed`/`unsigned long`.

Um zu erfahren, wie groß eine `int` werden darf, kann man also ein kleines Programm schreiben:

```
/* grintmax.c 30. 7.95 kw
 */

#include <limits.h>

main()
{
  printf( "int hat maximal den Wert %d\n", INT_MAX );
}
```

size_t ist ein ganzzahliger, vorzeichenloser Datentyp, der fast immer gleichbedeutend ist mit dem Typ `unsigned long int` oder mit `unsigned int`. Er dient dazu, die Größe eines Speicherbereichs anzugeben. Dazu kann je nach Rechner ein 2 oder 4 Byte großer Wert nötig sein, deshalb wurde der Typ `size_t` geschaffen, der nun auf jeder Implementation für das Arbeiten mit Speichergrößen verwendet werden kann. `size_t` ist in stddef.h, aber meist auch in anderen Headerdateien vereinbart.

8.4.2 Logischer Datentyp(*boolean, logical*)

existiert nicht. Einfach nicht da.

Stattdessen wird irgendein ganzzahliger oder gebrochener Datentyp verwendet. In der Regel ist das `int`, kann aber auch `char`, `long` oder gar ein *Zeiger* sein oder `double`. Die einzige Regel, die man dabei beachten muß: Der Zahlenwert `0` (oder bei `float` und `double`: `0.0`, bei *Zeigern*: `NULL`) zählt als *logisch falsch*, alles andere als *logisch wahr*[11].

Alle Vergleichsausdrücke liefern dementsprechend den Zahlenwert `1`, also *wahr*, wenn die entsprechende Bedingung erfüllt ist, oder `0`, also *falsch*, wenn die Bedingung nicht erfüllt ist.

[11] Da ich selbst ausgesprochen vergeßlich bin, war diese Regel bei meinen ersten Gehversuchen in *C* der häufigste Grund, in [K&R II] nachzuschlagen. Daraufhin notierte ich auf dem Einband „0 ist false, alles andere ist true“.
Um dem Leser —Ihnen!— diese Arbeit und meinem Werk die Verunstaltung zu ersparen, wurde dieser Satz gleich zum Titel erkoren.

Beispiel:

```
    x = 10;
    y = 20;
    if( x<y )
        printf( "x ist kleiner als y\n" );
    else
        printf( "x ist groesser oder gleich y\n" );
```

Da `x` tatsächlich kleiner ist als `y`, liefert der Ausdruck `x<y` das Ergebnis `1` (also *logisch wahr*) und die erste der beiden `printf`-Anweisungen wird ausgeführt.

Wäre `x` nicht kleiner als `y`, dann würde der Ausdruck `x<y` das Ergebnis `0` liefern.

So ersetzen in *C* normale (meist ganzzahlige) Datentypen die logischen Variablen aus anderen Sprachen.

Achtung! Wird eine Gleitkommazahl berechnet, und dann in einem logischen Ausdruck verwendet, wird sie fast immer als *ungleich 0* bewertet, da sie durch Rundungsfehler meistens nicht den Wert `0` trifft.

Deshalb sollte man statt

```
    double  a;
    /* ... */
    a = /* ... */;
    if( a ) /* ... */
```

lieber etwa folgendes schreiben:

```
#include <math.h>
#include <float.h>
/* ... */
    double  a;
    /* ... */
    a = /* ... */;
    if( fabs( a )>FLT_EPSILON ) /* ... */
```

8.4.3 Gebrochene Zahlen

An gebrochenen Zahlen (Gleitkommazahlen) kennt *C* die drei Datentypen `float`, `double` sowie `long double`. Mit dem Verhältnis dieser Typen untereinander ist es ähnlich bestellt wie mit `short`, `int` und `long`: Die Implementation muß sicherstellen, daß `double` größer oder gleich `float` ist und `long double` größer oder gleich `double` (bezogen auf Wertebereich und Genauigkeit). Tatsächlich sind `long double` und `double` oft das selbe.

Analog zu limits.h sind in float.h bei einer sauberen ANSI-Implementation die Wertebereiche und Genauigkeiten für die Gleitkommazahlen definiert:

- `FLT_RADIX` gibt die Basis der internen Darstellung des Gleitkommaexponenten an (meist 2).
- `FLT_ROUNDS` ist als 1 definiert, wenn zur nächstliegenden Zahl gerundet wird.
- `FLT_MANT_DIG`/`DBL_MANT_DIG`/`LDBL_MANT_DIG` geben die Stellenzahl für die Mantisse (zur Basis `FLT_RADIX`) für die Typen `float`/`double`/`long double` an.
- `FLT_EPSILON`/`DBL_EPSILON`/`LDBL_EPSILON` geben das kleinste `x` an, so daß: `1.0+x>1.0` gilt.
- `FLT_DIG`/`DBL_DIG`/`LDBL_DIG` (ungefähre Stellenzahl) in Dezimalziffern.
- `FLT_MIN_EXP`/`DBL_MIN_EXP`/`LDBL_MIN_EXP` kleinstes `x`, so daß 2^{x-1} normalisiert[12] ist.
- `FLT_MAX_EXP`/`DBL_MAX_EXP`/`LDBL_MAX_EXP` größtes `x`, so daß 2^{x-1} darstellbar ist.
- `FLT_MIN`/`DBL_MIN`/`LDBL_MIN` kleinste, normalisierte, positive Zahl.
- `FLT_MAX`/`DBL_MAX`/`LDBL_MAX` größte, darstellbare Zahl.
- `FLT_MIN_10_EXP`/`DBL_MIN_10_EXP`/`LDBL_MIN_10_EXP` kleinstes `x`, so daß 10^x noch normalisiert ist.
- `FLT_MAX_10_EXP`/`DBL_MAX_10_EXP`/`LDBL_MAX_10_EXP` größtes `x`, so daß 10^x noch darstellbar ist.

Die Zahlenbereiche des verwendeten Compilers erhält beispielsweise man so:

```
/* grfltmax.c 31 .7.95 kw
 */

#include <float.h>

main()
{
  printf( "Die groesste float-Zahl ist %12.4e.\n", FLT_MAX );
  printf( "double dagegen haelt bis    %12.4e mit!\n", DBL_MAX );
}
```

8.4.4 Zeiger

(oder auch *pointer*) ist ein Typ, der eine Adresse des Arbeitsspeichers aufnehmen kann, also beispielsweise die Adresse einer Variablen oder einer Funktion.

Ein Zeiger ist an einen anderen Typ gebunden, beispielsweise ein Zeiger auf eine `int` oder ein Zeiger auf eine `double`.

Beschrieben wird ein Zeigertyp durch den entsprechenden daran gebundenen Datentyp mit einem nachgestellten Stern `*`. Ein Zeiger auf eine `int` hat also den Typ

[12] Normalisierte Gleitkommazahlen sind solche, deren binäre Mantisse mit einer 1 beginnt und die dadurch die volle Genauigkeit der Mantisse ausnutzt. Manche Rechnersysteme verwenden nur normalisierte Zahlen. Andere können durch Verwendung von „denormalisierten“ Zahlen auch noch Werte darstellen, die enger an der 0 liegen. Dann geht aber Genauigkeit verloren. Eine detaillierte Beschreibung ist in [DEK II] zu finden.

int *, ein Zeiger auf eine double ist double *, ein Zeiger auf eine char ist char * und so weiter.

Ein Zeiger kann auch auf eine Variable vom Typ Zeiger verweisen und erhält dann als Typbezeichnung den Typ, auf den verwiesen wird (z.B. char *) mit einem nachgestellten *, also insgesamt zwei Sterne (char **). Eine Variable vom Typ char ** zeigt also auf eine andere Variable, die ebenfalls ein Zeiger ist und wiederum auf eine Variable oder Konstante vom Typ char verweisen kann.

Hat man eine Variable vom Typ Zeiger mit dem Namen p, dann meint man mit p den Inhalt der Zeigervariablen, also die Speicheradresse. Mit *p bezeichnet man dagegen das, worauf die Zeigervariable verweist.

Beispiel:

```
int *p;        /* p ist eine Variable vom Typ "Zeiger auf int"
                */
int i, j;      /* i und j sind "normale" int-Variablen
                */

/* Bisher zeigt p auf irgendwas, aber keiner weiss, worauf.
 * Deshalb erst mal auf i zeigen lassen:
 */
p = &i;        /* i ist die Variable, &i ihre Adresse zur
                * Laufzeit des Programms im Speicher
                * p ist die Zeigervariable, die die Adresse
                * einer int aufnehmen kann und hier auch tut.
                */

i = 25;        /* i erhaelt einen Zahlenwert.
                */
j = *p;        /* j erhaelt den gleichen Zahlenwert, denn:
                * In i ist die Zahl 25 gespeichert.
                * in p ist die Adresse von i hinterlegt und
                * *p meint nicht diese Adresse, sondern das,
                * worauf die Adresse zeigt, also den Inhalt von
                * i und damit den Wert 25.
                */

p = &j;        /* p zeigt jetzt nicht mehr auf i, sondern auf j.
                */
*p = 30;       /* An die Adresse, auf die p zeigt (*p) wird die
                * Zahl 30 geschrieben. Da p auf j zeigt, hat j
                * jetzt den Wert 30. i wird dabei nicht
                * veraendert.
                */
```

Der Zustand vor der Zuweisung p = &i; ist in Bild 1 links zu sehen, der Zustand danach im selben Bild rechts. Dabei wird angenommen, daß

- der Datentyp int 2 Byte belegt und ein Zeiger 4 Byte
- die Variablen p, i und j ab der fiktiven Adresse 1100 hintereinander im Speicher liegen (zur Laufzeit des Programms)

Die Diagramme stellen einen möglichen Ausschnitt des Arbeitsspeichers zur Laufzeit dar. Im allgemeinen Fall befinden sich die Variablen natürlich an anderen Adressen, und brauchen gar nicht hintereinander zu liegen.

In der Speicherzelle selbst ist der Inhalt eingetragen, daneben die (angenommene) Adresse und der jeweilige Variablenname. Ein Fragezeichen in der Speicherzelle bedeutet, daß der Inhalt nicht bekannt ist, sondern einen zufälligen Wert hat.

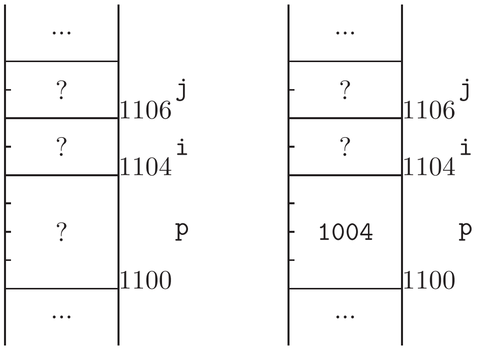


Abbildung 1: Vor und nach der Zuweisung `p = &i;`

Das Bild 2 zeigt die Situation nach der Zuweisung `i = 25;` (links) bzw. `j = *p;` (rechts).

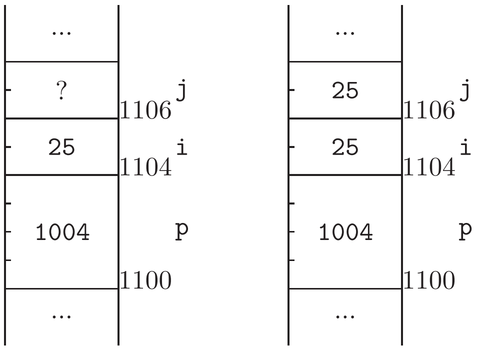


Abbildung 2: Nach `i = 25;` bzw. `j = *p;`

Das Resultat der Zuweisungen `p = &j;` (links) bzw. `*p = 30;` (rechts) ist in Bild 3 skizziert.

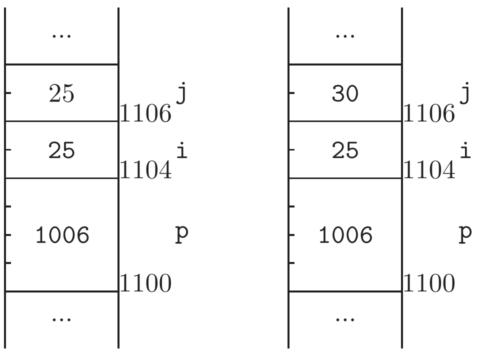


Abbildung 3: Nach `p = &j;` bzw. `*p = 30;`

Zeiger auf Funktionen: Zeiger können auch auf Funktionen zeigen. Die Funktion kann dann über den Zeiger aufgerufen werden.

Beispiel:

```
/* grfunzei.c 31. 7.95 kw
 *
 * testfunktion() gibt den uebergebenen Wert aus.
 */

void testfunktion( int wert )
{
  printf( "%d\n", wert );
}

int main()
{
  void (*fun_zeiger)( int wert );

  /* direkter Aufruf der Testfunktion:#
   */
  testfunktion( 5 );

  /* Holen der Adresse der Funktion:
   */
  fun_zeiger = &testfunktion;

  /* Indirekter Aufruf:
   */
  (*fun_zeiger)( 6 );
}
```

Dieses ansonsten sinnlose Beispiel zeigt die Verwendung eines Zeigers auf eine Funktion.

In `main()` wird zuerst eine Variable vereinbart:

```
void (*fun_zeiger)( int wert );
```

definiert eine Variable mit dem Namen `fun_zeiger`. Diese Variable hat den Typ „Zeiger auf eine Funktion, die den Rückgabewert `void` hat und einen Parameter vom Typ `int`“. In `fun_zeiger` kann man also die Adresse einer Funktion speichern.

```
    testfunktion( 5 );
```

ruft eine solche Funktion mit dem Argument 5 direkt auf; die Funktion gibt die Zahl 5 auf dem Bildschirm aus.

```
    fun_zeiger = &testfunktion;
```

Diese Zeile speichert in der Variablen `fun_zeiger` die Adresse der Funktion `testfunktion()`.

Diese Schreibweise funktioniert bei älteren und neueren *C*-Compilern. Nach dem (neueren) ANSI-Standard kann der Adreßoperator `&` auch entfallen, da mit dem Namen einer Funktion gleichzeitig ihre Adresse gemeint ist. Dieses Verhalten der neueren Compiler ist mit der Behandlung von Feldern (als Adresse des ersten Feldelements) vergleichbar, siehe dazu **8.8.1 Felder** auf Seite 61.

```
(*fun_zeiger)( 6 );
```

Damit wird die Funktion, auf welche `fun_zeiger` zeigt, mit dem Argument 6 aufgerufen. Da in `fun_zeiger` die Adresse von `testfunktion()` gespeichert ist, wird diese Funktion aufgerufen.

Da bei *ANSI-C*-Compilern wie gesagt der Funktionsname und die Adresse der Funktion gleich behandelt werden, könnte man diese Zeile auch einfacher schreiben:

```
fun_zeiger( 6 );
```

Zur Verdeutlichung, daß mit `fun_zeiger` tatsächlich über einen vereinbarten Zeiger auf eine Funktion zugegriffen wird, könnte man trotzdem die alte Schreibweise beibehalten.

Zeiger auf Funktionen werden meistens verwendet, um Funktionen als Parameter an andere Funktionen übergeben zu können. Ein Beispiel dafür ist die Übergabe der Vergleichsfunktion an `bsearch()` (Seite 181) und `qsort()` (Seite 213) aus der Standardbibliothek.

Eine andere Anwendungsmöglichkeit ergibt sich, wenn in Abhängigkeit von einem ganzzahligen Ausdruck eine Funktion aus einer Menge von Funktionen aufgerufen werden muß. Dann kann man ein Feld von Zeigern auf Funktionen definieren und mit den Adressen der Funktionen füllen. Die jeweilige Funktion ruft man aus dem Feld auf (mit dem Ausdruck als Index). Das ist beliebt, weil es sehr schnelle Programme ergibt.

void *: Es gibt aber nicht nur Zeiger auf bestimmte Datentypen, sondern auch einen „Jokerzeiger“, der auf jede beliebige Variable oder andere Speicherstelle zeigen darf. Dies ist der Datentyp `void *`.

Vorsicht: Vereinbart man eine Zeigervariable, dann heißt das nicht, daß diese Variable auf ein entsprechendes Element zeigt! Vielmehr bedeutet die Vereinbarung nur, daß die Variable auf ein Element zeigen *kann*. Der Inhalt der Zeigervariablen ist ebenso wie bei anderen Variablen durch die Typvereinbarung noch nicht gegeben. Erst durch Initialisierung oder Zuweisung einer Adresse (einer Variable oder mit `malloc()` beschaffter freier Speicher) zeigt die vereinbarte Zeigervariable auf einen entsprechenden Wert.

Ungültiger Zeiger: Eine besondere Konstante existiert vom Typ `void *`, also ein Zeiger auf einen beliebigen Typ. Diese Konstante hat den Namen `NULL` und ist immer in stddef.h vereinbart. Ein Zeiger mit diesem Wert kann niemals auf gültige Daten zeigen und wird deshalb als Wert für ungültige Zeiger verwendet. Beispielsweise liefert **malloc()** (Seite 206) wenn es einen Speicherblock beschaffen soll, einen Zeiger auf den beschafften Speicher oder (wenn kein Speicher beschafft werden kann) den Wert `NULL`. Wenn man den Rückgabewert von `malloc()` also mit `NULL` vergleicht, dann kann man erkennen, ob der Funktionsaufruf erfolgreich war.

Der Umkehrschluß gilt leider nicht: Wenn ein Zeiger einen anderen Wert als `NULL` hat, dann kann man daraus noch nicht schließen, daß der Zeigerwert gültig ist!

Auf jedem System wird ein Speicherzugriff auf die Adresse `NULL` zu einem Laufzeitfehler führen. Zugriffe auf alle anderen Adressen sind Glückssache, wenn ein Zeiger einen unsinnigen Inhalt hat: es kann ein Laufzeitfehler auftreten, möglicherweise läuft das Programm aber auch mit fehlerhaften Daten weiter.

Deshalb die dringende Empfehlung: An einen Zeiger, der nicht definitiv auf eine gewollte Adresse zeigt, immer `NULL` zuweisen! Dann führt eine versehentliche Verwendung wenigstens zuverlässig zu einem Programmabbruch.

Ebenso gefährlich ist es, Zeiger auf lokale Variablen zu verwenden, wenn diese gar nicht mehr existieren. Beispiel:

```
/* zeiglok.c 31. 7.95 kw
 */

/* Diese Funktion liefert einen Zeiger auf eine lokale Variable.
 * Das ist ausgemachter Schwachsinn!
 * Bitte nie nachahmen!
 */
int *f_p( void )
{
  int lokal;
  return &lokal;
}

int main( int nargs, char *args[] )
{
  int *ptr;

  ptr = f_p();

  /* ptr zeigt jetzt auf eine gar nicht mehr existierende
   * Variable. Der folgende Aufruf ist also kriminell:
   */
  *ptr = 25;
  printf( "Die Variable wurde auf 25 gesetzt.\n" );
  printf( "Ihr Wert ist jetzt %d\n", *ptr );

  return 0;
}
```

Das genaue Verhalten eines solchen Programms ist nicht vorhersehbar; bei mir entsteht folgende Ausgabe:

```
Die Variable wurde auf 25 gesetzt.
Ihr Wert ist jetzt 25
```

Solche Fehler können nicht vom Compiler erkannt werden und sind unter Umständen auch vom Programmierer schwer zu entdecken.

Zeigerarithmetik: Mit Zeigern kann man auch bedingt rechnen. Dies ist allerdings auf „sinnvolle“ Operationen beschränkt. Multiplikationen, Divisionen, Modulobildung und ähnliches ist verboten.

Erlaubt sind dagegen Addieren und Subtrahieren von ganzzahligen Ausdrücken (dies ergibt ein Ergebnis vom Typ des zugrundeliegenden Zeigers) sowie das Subtrahieren zweier Zeiger (dies ergibt ein Ergebnis vom ganzzahligen Typ `ptrdiff_t`).

Beim Subtrahieren zweier Zeiger müssen beide auf ein Objekt des gleichen Typs zeigen.

Bei Zeigerarithmetik ist es sehr wichtig zu wissen, daß der beteiligte ganzzahlige Datentyp (also das Ergebnis beim Subtrahieren zweier Zeiger oder der eine Operand beim Addieren zu einem Zeiger) nicht in Byte gemessen wird. Bemessungsgröße ist vielmehr die Größe, die ein Objekt des Typs im Speicher einnimmt, auf das die Zeiger verweisen.

Da der Compiler dazu die Größe des Objekts kennen muß, kann man mit Zeigern vom Typ `void *` keine Arithmetik treiben!

Beispiel:

```
double      f[10];       /* Feld mit 10 Elementen             */
double     *ptr;         /* Zeiger auf double                 */

ptr = &f[0];             /* ptr zeigt auf erstes Feldelement. */
*ptr = 0.0;              /* Dahin eine 0.0 schreiben.         */
*(ptr+2) = 3.1415;       /* f[2] wird belegt.                 */
                         /* (ptr+2) zeigt nicht 2 Byte hinter */
                         /* ptr, sondern 2 double-Elemente    */
                         /* hinter ptr. Bei 8 Byte je double  */
                         /* sind das 16 Byte hinter ptr.      */
```

Analog liefert die Subtraktion zweier Zeiger nicht den Abstand der beiden Operanden in Byte gemessen, sondern gemessen in der Größe des jeweiligen Objekts, auf das verwiesen wird.

Dadurch ist Zeigerarithmetik unabhängig vom verwendeten Rechnersystem und den darauf verwendeten Größen. Die Programme bleiben also selbst bei diesen Rechnereien portabel, weil die Rechnerabhängigkeiten nicht im Quelltext berücksichtigt werden, sondern beim Übersetzen vom jeweiligen Compiler.

C erhält durch Zeigerarithmetik erst richtig Leben; die größere Effektivität von Programmen in *C* rührt zu einem großen Teil davon. Angenommen, eine Funktion soll ein Feld mit `double`-Elementen löschen, also alle Werte zu 0.0 setzen. Ein ehemaliger *FORTRAN*- oder *Pascal*-Programmierer würde die Funktion vielleicht so schreiben:

```
/* grloel.c 31. 7.95 kw
 */

#include <stddef.h>

void loesche_double( double feld[], size_t wieviele )
{
  size_t      i;

  for( i=0; i<wieviele; i++ )
    feld[i] = 0.0;
}
```

Das funktioniert natürlich, aber die Lösung enthält überflüssige Berechnungen. Abgesehen von der nötigen Erhöhung des Schleifenzählers (Auswertung von `++`) muß bei jedem Durchlauf die Adresse des Elementes `feld[i]` berechnet werden. Dazu muß das Programm die Adresse hernehmen, die mit dem Namen `feld` verbunden ist (das ist die Adresse des ersten Feldelements), und dazu ein Produkt addieren, nämlich `i*sizeof(double)` (zur Auswertung von `feld[i]`).

Mit Zeigerarithmetik läßt sich die Funktion wesentlich schneller machen:

```
/* grloes.c 31. 7.95 kw
 */

#include <stddef.h>

void loesche_double( double *feld, size_t wieviele )
{
  double
    *pfeld = feld,
    *pende = feld+wieviele;

  while( pfeld<pende )
    *pfeld++ = 0.0;
}
```

Jetzt wird der Parameter `feld` als Zeiger auf das erste Element aufgefaßt. Ein Zeiger (`pfeld`) wandert durch das Feld und zeigt nacheinander auf die zu löschenden Elemente. Neben der eigentlichen Zuweisung (`=`) muß in der Schleife nur noch zu `pfeld` bei jedem Durchlauf der Wert `sizeof(double)` addiert werden (Auswertung von `++`). Die aufwendige Multiplikation entfällt, außerdem hat man statt zwei Additionen nur noch eine.

ptrdiff_t dient analog zu `size_t` dazu, die Differenz von 2 Zeigern aufzunehmen. Meist wie `size_t`, aber mit Vorzeichen; also `int` oder `long int`.

8.5 Freier Speicher

Zusätzlich zu den fest vereinbarten Variablen kann ein Programm zur Laufzeit bei Bedarf Speicher anfordern (*allokieren*), in dem es ebenfalls Daten unterbringen kann. Siehe dazu die Funktionen **malloc()** (Seite 206), **calloc()** (Seite 184), **realloc()** (Seite 216), und **free()** (Seite 195).

Beim Aufruf von `malloc()` gibt man die gewünschte Speichermenge an und erhält einen Zeiger auf einen entsprechend großen freien Speicher. Diesen Speicher kann man nur über Zeiger beschreiben und lesen, aber ansonsten wie eine Variable behandeln. Ist beim Aufruf von `malloc()` kein ausreichend großer Speicherblock frei, dann liefert `malloc()` den ungültigen Zeiger `NULL` zurück.

Die gewünschte Speichermenge muß man in der Einheit Byte angeben. Wie erfährt man jetzt aber, wieviele Byte man für beispielsweise 10 Elemente vom Typ `int` benötigt? Je nach Rechner wird eine `int` in 2, 4 oder 8 Byte gespeichert. Dementsprechend müßte man `malloc(20)`, `malloc(40)` oder `malloc(80)` aufrufen, um die richtige Speichermenge zu erhalten. Um dieses Problem rechnerunabhängig zu lösen, gibt es den Operator `sizeof`. Dahinter kann man in Klammern einen Datentyp oder (mit oder ohne Klammern) eine Variable eintragen und erhält daraus die Größe des entsprechenden Objekts in Byte. Mit `malloc( 10*sizeof(int) )` übergibt man als Argument an `malloc()` auf jedem Rechner die Anzahl Byte, die jeweils für 10 Werte vom Typ `int` benötigt werden.

`sizeof` ist keine Funktion, obwohl die Verwendung mit Klammern ähnlich aussieht. Vielmehr wird bereits bei der Übersetzung vom Compiler für den ganzen Ausdruck eine Konstante eingesetzt.

Mit `free()` kann man einen mit `malloc()` (oder einer der anderen Funktionen) allokierten Speicherblock wieder freigeben, wenn man ihn nicht mehr benötigt.

Beispiel:

```
/* damalloc.c 30. 7.95 kw
 * Beispiel fuer malloc().
 */

/* Mit #include wird an dieser Stelle der Inhalt der Datei
 * stdlib.h eingefuegt. Dies deshalb, weil darin die
 * Funktion malloc() vereinbart ist.
 */

#include <stdlib.h>

main()
{
  double *feld;   /* Vereinbarung der Variablen feld.
                   * feld hat nicht den Typ double
                   * (doppelt genaue Gleitkommazahl),
                   * sondern double* (Zeiger auf eine
                   * Gleitkommazahl), kann also die
                   * Adresse einer double speichern.
                   */

  /* mit malloc() Platz fuer 10 double-Elemente holen:
   * malloc() reserviert in einem unbenutzten
   * Speicherbereich des Rechners Platz fuer 10 Werte
   * der Groesse, die double benoetigt.
   * Wenn genug Platz frei ist, dann liefert malloc()
   * die Adresse des Anfangs des Speicherbereichs.
   * Wenn kein ausreichend grosser Bereich frei ist,
   * liefert malloc() statt einer gueltigen Adresse
   * den Wert NULL (vereinbart in stdlib.h).
   * Die von malloc() gelieferte Adresse wird in der
   * Variablen feld gespeichert, die ja eine Adresse
   * aufnehmen kann.
   */
  feld = malloc( 10*sizeof(double) );

  /* ist die gelieferte Adresse sinnvoll, also ungleich NULL?
   */
  if( feld!=NULL )
  {
    /* hier kann man feld[0] bis feld[9] verwenden
     * ...
     */
    free( feld );  /* Speicher wieder freigeben         */
  }
  else
    puts( "kein Speicher frei!" );       /* Fehler         */
}
```

8.6 Attributangaben

Mit Attributangaben kann man Definitionen und Deklarationen näher spezifizieren.

`extern` sagt, daß die damit vereinbarten Namen nicht in dieser Quelltextdatei, sondern in einer anderen definiert werden. Die Vereinbarung ist damit nur eine Deklaration; sonst wäre sie eine Definition (falls es um die Deklaration von Variablen oder Konstanten geht; die Deklaration von Funktionen wird immer als `extern` behandelt, auch wenn es nicht explizit angegeben wird).

`static` hat zwei verschiedene Bedeutungen; je nachdem, ob es sich auf einen lokalen oder globalen Namen bezieht. Steht es außerhalb einer Funktionsdefinition, ist die Vereinbarung also global, dann darf der Linker einen als `static` bezeichneten Namen nicht nach außen an andere Dateien weitergeben; die damit definierten Namen sind also auf die eine Datei beschränkt, in der sie definiert werden. Für lokale Namen ist das ohnehin der Fall.

```
int         a, b;
static int  i, j;
```

vereinbart vier Variablen vom Typ `int`. Auf `a` und `b` kann man sowohl in dieser Datei zugreifen als auch mit:

```
extern    int a, b;
/* ... */
a = 25;
```

in anderen Dateien. `i` und `j` dagegen sind durch das Schlüsselwort `static` zur Geheimsache erklärt und von anderen Quelltexten aus nicht zugänglich. Dies wird auch häufig für Funktionen verwendet, die dann nur innerhalb der Datei aufrufbar sind, in der sie auch definiert werden.

Bei lokalen Vereinbarungen dagegen wirkt `static` anders: Normalerweise sind lokale Variablen nur gültig, solange der umgebende Block, also meist die umgebende Funktion, aktiv ist. Beim Verlassen des Blocks werden die Werte der Variablen ungültig und können beim nächsten Eintritt in den Block ganz andere Werte haben. Mit `static` wird dies verhindert. Damit vereinbarte Variablen behalten ihren Wert auch beim Verlassen der umgebenden Funktion und haben beim nächsten Aufruf den alten Wert. Bei rekursiv aufgerufenen Funktionen kann das ein Schuß nach hinten sein: jede Aufrufebene der Funktion arbeitet mit derselben Variable, wenn sie als `static` vereinbart ist! Sonst erhält jede Aufrufebene einer Funktion einen eigenen Satz von Variablen, so daß Änderungen einer Variable in einer Aufrufebene die Werte in den anderen Ebenen nicht beeinflussen. Bei `static`-Variablen ist das nicht der Fall; hier gibt es die betreffende Variable nur einmal für alle Aufrufebenen. Siehe dazu auch **13 Rekursion** auf Seite 95.

Das Gegenteil von `static` ist das Schlüsselwort `auto`. Dieser Zusatz sagt dem Compiler, daß die so vereinbarte Variable nur lokal und temporär, also nicht `static`, im Speicher gehalten werden soll (automatische Variable). Für statische globale Variablen ist dieser Zusatz nicht zulässig, und bei lokalen Variablen ohnehin Standard. Den Zusatz `auto` kann man also auch immer genauso gut weglassen.

`register` ist ein Hinweis des Programmierers an den Compiler, daß eine Variable besser in einem Register aufgehoben werden sollte als im normalen Arbeitsspeicher. Der Zugriff auf Register ist bei den meisten Rechnern erheblich schneller als auf normale Variablen. Der Compiler darf den Hinweis auch ignorieren. Da die Anzahl an Registern in der Regel sehr begrenzt ist (keine bis maximal ein Dutzend

Variablen kann man in Registern unterbringen), muß der Compiler ja die Möglichkeit haben, überzählige Variablen anderswo unterzubringen. Beispielsweise braucht man gar keine Felder in Registern unterbringen; dafür ist nie Platz. Zudem darf man auf Registervariablen nicht mit dem Adreßoperator `&` zugreifen, da sie ja eventuell gar keine Adresse im Speicher haben. Lohnen kann sich `register` höchstens bei Schleifenzählern oder Variablen, die in kleinen Schleifen viel geändert werden. Gute Compiler erkennen das aber oft auch schon selbst.

Beispiel:

```
register int   i;
register long  l;
    /* ... */
    while( --i ) l += i;
```

Abgesehen vom nicht mehr zulässigen Adreßoperator (`&`) ändert sich durch `register` nichts am Programm; wenn man Glück hat, läuft es dann aber schneller[13].

Die drei Speicherklassen `static`, `auto` und `register` schließen sich gegenseitig aus.

Mit `const` sagt man dem Compiler, daß man eine Variable nicht vom Programm aus ändern will. Die einzige legale Möglichkeit, einer `const`-Variablen einen Wert zu geben, ist die Initialisierung[14]. Macht man an eine solche Variable trotzdem eine Zuweisung, dann erkennt der Compiler dies (hoffentlich) und gibt eine Fehlermeldung aus. Dies ist als zusätzliche Fehlerkontrolle oft sinnvoll. Formal sind so vereinbarte Variablen aber immer noch Variablen! Siehe dazu auch `volatile` im nächsten Abschnitt.

Mit

```
const    int zwanzig = 20;
```

kann man die Variable `zwanzig` nur lesend verwenden, aber eine Zuweisung wie etwa:

```
   zwanzig = 25;
```

ist dann nicht möglich.

Mit

```
const char *p = "abc123";
```

bezieht sich `const` nicht auf den Zeiger `p`, sondern besagt, daß das, worauf `p` zeigt, nicht geändert werden darf.

Danach ist also

```
p++;
```

zulässig; nicht aber

```
*p = 'e';
```

[13] Tatsächlich hat dies kaum noch Bedeutung, weil moderne Compiler so gut optimieren, daß Hinweise des Programmierers wie `register` ohnehin ignoriert werden.

[14] Eine illegale Möglichkeit wäre, einer Funktion die Adresse der Variable zu übergeben, wodurch die Funktion den Wert über einen Zeiger ändern kann. Beispielsweise könnte `scanf()` so eine `const`-Variable ändern. Dies muß aber nicht auf allen Systemen funktionieren!

Will man einen Zeiger selbst als const erklären, dann muß man ihn so vereinbaren:

```
char   const *p = "einszweidrei";
```

Dann ist der Zeiger selbst konstant.

Beides gleichzeitig ist natürlich auch erlaubt (aber recht selten):

```
const char const *p = "appel und en ei";
```

Damit ist der Zeiger p konstant und darf nicht geändert werden; ebenso das worauf er zeigt.

In gewisser Hinsicht das Gegenteil von const ist volatile. Damit gibt man dem Compiler an, daß sich ein Wert eventuell ändern kann, ohne daß das Programm darauf Einfluß hat. Dies ist bei Systemvariablen des Rechners manchmal der Fall. Wenn man dem Compiler dann keinen Hinweis darauf geben kann, ist er eventuell zu schlau. Angenommen, an der Adresse 456 soll bei unserem Rechner in irgendeiner Form die Systemzeit hinterlegt sein und vom Rechner selbst laufend angepaßt werden. Dann kann man so einen Zeiger auf diese Adresse richten und verwenden:

```
unsigned long    *systemzeit_p = (unsigned long *)456;
long              i;
unsigned long     wert;
    /* ... */
    for( i=0; i<500000; i++ )
        {
            wert = *systemzeit_p;
            /* ... */
        }
```

Der Compiler könnte jetzt auf die Idee kommen, daß in der for-Schleife ja ohnehin immer der gleiche Wert an wert zugewiesen wird und das Programm so umbauen:

```
unsigned long    *systemzeit_p = (unsigned long *)456;
long              i;
unsigned long     wert;
    /* ... */
    wert = *systemzeit_p;
    for( i=0; i<500000; i++ )
        {
            /* ... */
        }
```

Dies ist eigentlich eine hervorragende Idee und wirklich gute Compiler machen das auch, da dann die Zuweisung anstatt 500000 mal in der Schleife nur einmal vor der Schleife ausgeführt wird. Der Compiler kann ja schließlich nicht wissen, daß der Wert, auf den systemzeit_p zeigt, laufend vom Rechner selbst geändert wird. Dadurch ist die eigentlich gut gemeinte Optimierung ein Eigentor.

Diesen Effekt kann man verhindern, indem man alle Werte, die von außerhalb des eigenen Programms geändert werden können, als `volatile` vereinbart:

```
volatile unsigned long  *systemzeit_p = (unsigned long *)456;
long                     i;
unsigned long            wert;
    /* ... */
    for( i=0; i<500000; i++ )
        {
            wert = *systemzeit_p;
            /* ... */
        }
```

So werden zu dreiste Programmoptimierungen durch den Compiler vermieden.

`const` und `volatile` können auch kombiniert werden: `const volatile` bedeutet, daß eine Variable vom Programm aus nicht geändert werden darf (`const`), wohl aber eventuell von außerhalb geändert werden kann (`volatile`). Damit ist es dem Programmierer untersagt, den Wert vom Programm aus zu ändern; aber der Compiler rechnet damit, daß dies von außerhalb geschehen kann.

8.7 struct und union

8.7.1 Strukturen

Sinn von Strukturen ist es, logisch zusammengehörende Informationen auch von der Programmierung her zusammenzufassen.

Mit dem Schlüsselwort `struct` kann man Datentypen schaffen, die es bislang nicht gibt und die aus einfacheren Typen zusammengesetzt sind.

Die Definition solcher Strukturen entspricht der Vereinbarung von `RECORD`s in *Pascal* oder *Modula2*.

Einfache Variablen kann man etwa so vereinbaren:

```
int     i, j;
```

Entsprechend werden Strukturen vereinbart:

```
struct
{
    int
        anzahl,
        farbe;
    double
        wert;
}
a, b;
```

deklariert 2 Variablen mit den Namen `a` und `b`. Jede dieser beiden Variablen enthält zwei ganze Zahlen (`anzahl`, `farbe`) und eine doppelt genaue Gleitkommazahl (`wert`).

Die einzelnen Elemente kann man mit dem Variablennamen, einem Punkt (.) und dem Elementnamen ansprechen:

```
    a.anzahl = 5;
    a.farbe  = 0;
    a.wert   = 3.14159265;
    b = a;
    printf( "b = %d, %d, %f\n", b.anzahl, b.farbe, b.wert );
```

Braucht man eine bestimmte Struktur an mehreren Stellen, beispielsweise zur Vereinbarung eines Funktionsparameters, dann kann man auch die Struktur mit einem Strukturnamen (hier s) versehen und für spätere Vereinbarungen nutzen:

```
struct s
{
    int
        anzahl,
        farbe;
    double
        wert;
};

struct s  a, b;

    /* ... */
    a.anzahl = 5;
    a.farbe  = 0;
    a.wert   = 3.14159265;
    b = a;
    printf( "b = %d, %d, %f\n", a.anzahl, a.farbe, a.wert );
```

Mischformen sind auch möglich. Man darf Variablen einer bestimmten Struktur vereinbaren und gleichzeitig einen Strukturnamen vergeben, der später verwendet werden kann:

```
struct s
{
    int
        anzahl,
        farbe;
    double
        wert;
}  a, b;

print_struct_s( struct s parameter )
{
    printf( "%d, %d, %f",
            parameter.anzahl,
            parameter.farbe,
            parameter.wert
        );
}

main()
```

```
{
    a.anzahl = 5;
    a.farbe  = 0;
    a.wert   = 3.14159265;
    b = a;
    printf( "b = " );
    print_struct_s( b );
    printf( "\n" );
}
```

Als Einträge in Strukturen sind alle bis dahin[15] bekannten Datentypen erlaubt, insbesondere auch andere Strukturen oder Zeiger auf Strukturen.

Man könnte also etwa eine Koordinate definieren, die aus der x- und der y-Position auf einer Zeichenfläche besteht und diese Struktur für weitere Typen verwenden, wie einen Punkt oder eine Linie:

```
struct koord
{
    int
        x,
        y;
};

struct punkt
{
    struct koord    position;
    int             farbe;
};

struct linie
{
    struct koord
        anfang,
        ende;
    int
        farbe;
};

struct linie    l;
struct punkt    p;

/* ... */
    l.anfang.x = 235;
    l.anfang.y = 0;
    l.ende.x   = 100;
    l.ende.y   = 150;
    l.farbe    = 12;

    p.position = l.anfang;
    p.farbe    = l.farbe;
```

[15] heißt: bis zu der aktuellen Stelle im Quelltext

Wen beim Benutzen der Strukturdefinition das ewige Schreiben des Schlüsselwortes `struct` nervt, der kann sich mit `typedef` für eine Struktur einen neuen Namen ausdenken und diesen verwenden:

```
typedef struct
{
    int
        x,
        y;
} koord_t;

typedef struct
{
    koord_t         position;
    int             farbe;
} punkt_t;

typedef struct
{
    koord_t         anfang, ende;
    int             farbe;
} linie_t;

linie_t         l;
punkt_t         p;


l.anfang.x = 235;
l.anfang.y = 0;
l.ende.x   = 100;
l.ende.y   = 150;
l.farbe    = 12;

/* ... */
    p.position = l.anfang;
    p.farbe    = l.farbe;
```

Der Anhang `_t` an den Typnamen soll nur kennzeichnen, daß sich hinter diesen Namen ein Datentyp verbirgt und ist nicht zwingend. Etwas Systematik bei der Vergabe von Namen schadet aber nie.

Beim Übergeben einer Struktur an eine Funktion als Parameter wird die Struktur wie andere Variablen auch als Wertparameter übergeben. Dies kann bei großen Strukturen uneffektiv sein. Dann sollte man stattdessen die Adresse der Struktur übergeben und den Parameter in der Funktion als Zeiger auf die jeweilige Struktur behandeln.

Die Größe einer Struktur im Speicher muß übrigens nicht die Summe der einzelnen Elemente sein. Viele Rechner können nämlich nicht alle Datentypen an beliebigen Adressen unterbringen. Je nach Prozessor kann es sein, daß alle Typen mit 2, 4, oder 8 Byte an geraden (oder ggf. auf durch 4 oder 8 teilbaren) Adressen liegen müssen. Hat man auf einem solchen Rechner (angenommen: `char` benötigt 1 Byte, `int` 2 Byte) folgende Variable `s` vereinbart:

```
struct
```

```
{
    int     i1;
    char    c;
    int     i2;
}
    s;
```

dann muß der Compiler zwischen den Elementen `c` und `i2` ein ungenutztes Byte einfügen, damit `i2` auf einer geraden Adresse liegt.

Durch die Forderung nach geraden Adressen vergrößert sich also auf dem Beispielrechner die `struct` von 5 auf 6 Byte. Am besten ist es ohnehin, über die genaue Größe von irgendwelchen Objekten im Speicher keine Vermutungen anzustellen, sondern mit `sizeof` das Rechnen auf den Compiler abzuwälzen. In dem obigen Beispiel würde also `sizeof(s)` auf jedem Rechner die richtige Größe liefern.

Bitfelder sind spezielle Einträge in Strukturen. Sie haben mit den bisher bekannten Feldern[16] nichts zu tun!

Sinn derselben ist es, in Strukturen Platz zu sparen. Hat man beispielsweise ein System, auf dem eine `int`-Variable 16 bit, also 2 Byte, belegt, dann benötigt man für jeden `int`-Eintrag in einer `struct` natürlich auch 2 Byte. Oft braucht man aber nicht den ganzen Zahlenbereich, den der Typ `int` bietet. `char` trifft aber den benötigten Platz auch nicht immer genau.

Mit Hilfe von Bitfeldern kann man bei der Definition einer `struct` angeben, wieviele bits man für den entsprechenden Eintrag braucht. Dies ist aber nur bei Einträgen möglich, die als `int` deklariert werden (`signed` oder `unsigned`). Folglich darf die angegebene bit-Zahl nicht größer sein, als die Länge einer `int` auf dem verwendeten Rechnersystem.

Die bit-Zahl wird hinter dem Elementnamen und einem Doppelpunkt eingetragen. Der Zugriff auf die Elemente ist identisch mit den normalen Strukturelementen. Jedes Bitfeld kann wie eine andere ganze Zahl verwendet werden; nur ist ihr Wertebereich kleiner.

Zumindest hintereinander liegende Bitfelder in einer Struktur faßt der Compiler nach Möglichkeit in einem Maschinenwort zusammen, also in einer `int`.

Werden mehr Bitfelder angegeben, als in ein Maschinenwort passen, dann beginnt der Compiler automatisch ein neues Wort.

Das Bilden von Feldern aus Bitfeldern ist nicht zulässig, wohl aber von Feldern aus Strukturen mit Bitfeldern.

Das Berechnen der Adresse eines Bitfelds mit dem Adreßoperator `&` ist nicht zulässig.

Der Zugriff auf Bitfelder ist in der Regel langsamer als auf normale `struct`-Elemente.

Bitfelder kann man auch nutzen, um gezielt auf Teile eines Wortes zugreifen zu können, in dem einzelne bits oder Gruppen davon eine bestimmte Bedeutung haben. Dies ist jedoch implementierungsabhängig. Insbesondere ist im Standard auch nicht festgelegt, in welcher Reihenfolge die Teile im Wort liegen.

Beispiel:

[16] In der angelsächsischen Literatur wird das bisher bekannte Feld (zum Beispiel `int f[10];`) als „array“ bezeichnet, das hier vorgestellte Bitfeld dagegen als „bit field“. Zur Unterscheidung übersetzen einige Autoren den Begriff „array“ mit „Vektor“, was aber erstens nicht geläufig (da Bitfelder eine Eigenart von *C* sind) und zweitens bei mehrdimensionalen Feldern unpassend ist.

```
/* stpack.c 31. 7.95 kw
 */

/* Die folgende struct benoetigt
 * sizeof(double) + 3*sizeof(int) =
 * 8                3*2           =    (in Turbo-C, Prospero C)
 *                                = 14 Bytes.
 * bzw.
 * sizeof(double) + 3*sizeof(int) =
 * 8                3*4           =    (in GNU-C)
 *                                = 20 Bytes.
 */

typedef struct
{
  double          d;
  unsigned int    farbe;
  unsigned int    wert;
  unsigned int    an_aus;
}
test_normal;


/* Die folgende struct belegt nur 10 oder 12 Byte.
 * Wenn die angegebenen Wertebereiche ausreichen, dann kann man
 * diese struct genauso verwenden wie test_normal.
 */

typedef struct
{
  double          d;              /* 8 Bytes     */
  /* alles weitere passt in eine int, also
   * 16 oder 32 Bit, je nach Rechner:
   */
  unsigned int    farbe      :8;  /* 0...255     */
  int             wert       :4;  /* -8...7      */
  unsigned int    an_aus     :1;  /* 0...1       */
  /* die restlichen Bits der struct werden nicht
   * genutzt.
   */
}
test_packed;

main()
{
  test_packed      a;
  printf( "test_normal ist %ld Bytes lang\n",
          (long)sizeof(test_normal)
          );
  printf( "test_packed ist %ld Bytes lang\n",
          (long)sizeof(test_packed)
          );

  a.d      = 1.0;
```

```
  a.farbe  = 200;
  a.wert   = 5;
  a.an_aus = 1;
  printf( "a = %4.1f, %u, %d, %u\n",
          a.d, a.farbe, a.wert, a.an_aus
          );
}
```

Dieses Programm erzeugt auf meinem Rechner (Linux, mit GCC) die Ausgabe:

```
test_normal ist 20 Bytes lang
test_packed ist 12 Bytes lang
a =  1.0, 200, 5, 1
```

8.7.2 union

Eine `union` ist `struct` sehr ähnlich.

Der wesentliche Unterschied liegt darin, daß die Elemente einer `union` nicht hintereinander im Speicher liegen (wie bei einer `struct`), sondern alle an der gleichen Stelle im Arbeitsspeicher. Genauer gesagt: alle Elemente beginnen an der selben Stelle im Speicher (sind die Elemente unterschiedlich groß, dann enden sie dementsprechend an verschiedenen Stellen).

Das bedeutet, daß man immer nur eines der vereinbarten Elemente einer `union` belegen kann; alle anderen werden dabei überschrieben.

Eine `union` ist immer so groß wie ihr größtes Element.

Man kann eine `union` dazu benutzen, um unterschiedliche Elemente zu speichern, wenn man sich sicher ist, daß immer nur eine Variante benutzt wird.

Die Vereinbarung und Verwendung ist identisch mit der `struct`, es wird nur das Schlüsselwort `union` statt `struct` verwendet.

Beispiel:

```
/* stmanta.c 31. 7.95 kw
 */

union manta_oder_iq
{
  int     iq;
  float   geschwindigkeit;
};

typedef struct
{
  char                name[30];
  char                beruf[20];
  union manta_oder_iq miq;
}
person_t;

void lese_person( person_t *p )
{
  printf( "Bitte den Namen eingeben: " );
  scanf( "%s", (*p).name );
  printf( "Bitte den Beruf eingeben: " );
  scanf( "%s", (*p).beruf );
  if( strcmp( (*p).name, "Manni" ) )
  {
    printf( "Bitte den Intelligenzquotienten:  " );
    scanf( "%d", &((*p).miq.iq) );
  }
  else
  {
    printf( "Bitte die Hoechstgeschwindigkeit: " );
    scanf( "%f", &((*p).miq.geschwindigkeit) );
  }
}
```

```
void zeige_person( person_t p )
{
  printf( "Name: %s\nBeruf: %s\n", p.name,p.beruf );
  if( strcmp( p.name, "Manni" ) )
  {
    printf( "Der IQ ist %d\n", p.miq.iq );
  }
  else
  {
    printf( "Manni faehrt %f km/h\n", p.miq.geschwindigkeit );
  }
}

main()
{
  person_t    person;
  lese_person( &person );
  zeige_person( person );
  return 0;
}
```

Dieses Programm liest Name und Beruf einer Person ein. Wenn der Name „Manni“ ist, dann wird seine Höchstgeschwindigkeit abgefragt, ansonsten der IQ (Intelligenzquotient) der Person. `person.miq.iq` kann einen Namen speichern, oder `person.miq.geschwindigkeit` kann eine Geschwindigkeit speichern; nie aber beides gleichzeitig.

Die Ausgabe des Programms erfolgt analog: Am Namen wird erkannt, ob in `person.miq` eine Geschwindigkeit gespeichert ist oder ein IQ.

Die Verantwortung, aus einer `union` den Typ auszulesen, der auch darin gespeichert wurde, liegt beim Programmierer.

8.8 Felder und Strings

8.8.1 Felder

Muß man sich nicht einzelne Werte merken, sondern gleich mehrere gleichartige, dann bieten sich Felder an. Ein Feld wird oft auch *array* oder Vektor genannt.

Die Vereinbarung dafür ist auch nicht schwerer als für eine normale Variable:

`int f[10]` vereinbart ein Feld mit dem Namen `f`, welches 10 Elemente enthält. Jedes der 10 Elemente hat den Typ `int`.

In *C* kann man leider nicht die obere und untere Feldgrenze frei vorgeben wie in *Pascal* oder *FORTRAN*; in *C* hat das erste Feldelement immer den Index 0. `f` hat in unserem Beispiel 10 Elemente mit den Indices `0` bis `9`. Man kann also mit dieser Deklaration auf die Elemente `f[0]`, `f[1]`, `f[2]`, `f[3]`, `f[4]`, `f[5]`, `f[6]`, `f[7]`, `f[8]` und `f[9]` zugreifen. Auf das Element `f[10]` zuzugreifen ist schon nicht mehr zulässig, da nur 10 Elemente vereinbart sind.

Eine andere untere Feldgrenze als `0` ist in *C* nicht direkt möglich; ein möglicher Ausweg wird in **A.1 Vereinbarung von freien Feldgrenzen** auf Seite 241 gezeigt.

Mehrdimensionale Felder sind analog mit mehreren Dimensionsangaben möglich:

`int ff[2][3]` vereinbart ein Feld mit sechs Elementen: `ff[0][0]`, `ff[0][1]`, `ff[0][2]`, `ff[1][0]`, `ff[1][1]` und `ff[1][2]`.

Die Elemente stehen auch in der angegebenen Reihenfolge im Speicher, wenn das Programm ausgeführt wird. Dies ist die gleiche Reihenfolge wie in *Pascal*, aber anders als in *FORTRAN*.

Durchläuft man ein Feld in der abgespeicherten Reihenfolge, ändert sich also der letzte Index am schnellsten.

Das Wissen um die Reihenfolge der Elemente im Speicher ist zum einen wichtig, wenn man einzelne Elemente nicht nur über die Indices ansprechen will, sondern eine kreativere Adressierung mit Zeigerarithmetik benötigt.

Zum anderen kann es sich bei großen Feldern deutlich auf das Laufzeitverhalten auswirken: Durchläuft man ein mehrdimensionales Feld so, daß sich der letzte Index am schnellsten ändert, und der erste am seltensten, dann verwendet man aufeinanderfolgende Elemente. Dies wird auf modernen Systemen wesentlich schneller sein, als in großen Schritten mehrfach durch das Feld zu springen indem man den ersten Index am schnellsten ändert (erstens, weil der Prozessor durch *caching* und *prefetching* auf aufeinanderfolgende Elemente im Schnitt schneller zugreifen kann, und zweitens weil das Betriebssystem weniger Seiten ein- und auslagern muß, wenn man bevorzugt auf benachbarte Daten zugreift).

Damit können wir das Quadratprogramm aus **3 Kleine Beispiele** auf Seite 8 so ändern, daß zuerst alle Quadrate berechnet und in einem Feld gespeichert werden. Erst zum Schluß sollen alle berechneten Werte ausgegeben werden.

```
/* feq.c 31. 7.95 kw
 *
 * Dieses Programm gibt die Zahlen von 1 bis 10 und die Quadrate aus.
 */

#define IMAX        10

int printf( char *format, ... );   /* gibt format und eventuell
                                    * weitere Werte auf die Standard-
                                    * ausgabe.
                                    */

int main()
{
  int      i;                 /* eine ganzzahlige Variable als
                               * Schleifenzaehler vereinbaren.
                               */
  int      imalifeld[IMAX];   /* Feld fuer die Quadrate.
                               */


  printf( "Dieses Programm gibt die ersten %d Quadratzahlen",
          IMAX
          );
  printf( " auf dem Bildschirm aus.\n\n" );

  /* Schleife ueber alle zu berechnenden Zahlen:
   */
```

```
  for( i=1; i<=IMAX; i=i+1 )
  {
    imalifeld[i-1] = i*i;
  }

  /* Schleife ueber alle auszugebenden Zahlen:
   */
  for( i=1; i<=IMAX; i=i+1 )
  {
    printf( "Das Quadrat von %d ist %d.\n",
            i, imalifeld[i-1] );
  }

  return 0;                   /* und den Wert 0 zurueckgeben.
                               */
}
```

Die Verwendung von `i-1` als Index anstatt `i` ist deshalb nötig, weil `i` den Bereich von 1 bis 10 durchläuft, während im Feld `imalifeld` die Elemente von `0` bis `9` numeriert sind.

Ein Beispiel für ein mehrdimensionales Feld ist eine andere Version des Potenzenprogramms aus **3 Kleine Beispiele** auf Seite 8, das analog zum letzten arbeitet. Zuerst werden alle Potenzen berechnet und in einem zweidimensionalen Feld gespeichert. Dann werden alle Werte ausgegeben:

```
/* feqp.c 31 .7.95 kw
 */

/* Dieses Programm gibt die Zahlen von 1 bis 10 und die ersten 4

* Potenzen derselben aus.
                                            */

#define IMAX        10
#define PMAX        4

int printf( char *format, ... );   /* gibt format und eventuell
                                    * weitere Werte auf die Standard-
                                    * ausgabe.
                                    */

/* Diese Funktion liefert zu einem Wert die n-te Potenz:
 * n muss dabei groesser oder gleich 0 sein!
 */

long int potenz( long wert, int n )
{
  long    ergebnis;
  int     i;

  ergebnis = 1;
  for( i=1; i<=n; i=i+1 )
    ergebnis = ergebnis * wert;
```

```
  return ergebnis;
}

int main()
{
  int      i, p;  /* Schleifenzaehler fuer Zahl und Potenz
                   */

  /* In potenzfeld[i-1][p] soll die p-te Potenz zur Zahl i
   * stehen. Die erste Potenz einer Zahl ist die Zahl selbst,
   * die nullte Potenz einer Zahl ist immer 1!
   */
  long int potenzfeld[IMAX][PMAX+1];

  printf( "Dieses Programm gibt die ersten %d Potenzen zu den",
          PMAX
          );
  printf( " ersten %d Zahlen aus.\n\n",
          IMAX
          );

  /* Schleife ueber alle zu potenzierenden Zahlen:
   */
  for( i=1; i<=IMAX; i=i+1 )
  {
    /* Schleife ueber alle Potenzen zu einer Zahl:
     */
    for( p=0; p<=PMAX; p=p+1 )
    {
      /* die p-te Potenz zur Zahl i berechnen:
       */
      potenzfeld[i-1][p] = potenz( i, p );
    }
  }

  /* Schleife ueber alle auszugebenden Zahlen:
   */
  for( i=1; i<=IMAX; i=i+1 )
  {
    /* Schleife ueber alle Potenzen zu einer Zahl:
     */
    for( p=0; p<=PMAX; p=p+1 )
    {
      /* die p-te Potenz zur Zahl i ausgeben:
       */
      printf( "%2d^%-2d=%7ld;", i, p, potenzfeld[i-1][p] );
    }
    /* Nach den Potenzen zu einer Zahl ein Zeilenvorschub:
     */
    printf( "\n" );
  }

  return 0;                  /* und den Wert 0 zurueckgeben.
```

```
                                    */
}
```

Felder kann man von fast allen Datentypen bilden[17], also auch von zusammengesetzten Datentypen (Felder, Strukturen, `union`s) oder Zeigern.

Ein mehrdimensionales Feld (siehe oben) kann auch als Feld von Feldern betrachtet werden:

```
int f[2][3];
```

vereinbart ein Feld mit 2 Elementen; jedes dieser Elemente ist ein Feld mit 3 Elementen vom Typ `int`.

Die Übergabe von Feldern an Funktionen erfolgt anders als bei anderen Variablen.

Da Felder sehr groß werden können, wird automatisch statt des Feldes ein Zeiger auf das erste Feldelement übergeben.

Konsequenterweise wird in einer Funktion jeder als Feld deklarierte Parameter als Zeiger auf ein Feldelement behandelt. Dies fällt in der Regel nicht unangenehm auf. Zu beachten ist aber, daß dadurch Veränderungen an Feldelementen sich bis zum Aufrufer auswirken, was bei anderen Parametern durch den *call by value*-Aufruf nicht der Fall ist.

Zeiger auf Felder und normal vereinbarte Feldnamen können in den meisten Fällen identisch verwendet werden. Es gibt aber doch manchmal Ausnahmen!

Beispiel:

```
#include <stdlib.h>

int main( int nargs, char **args )
{
    int     feld1[10];
    int    *feld2;

    feld2 = malloc( 10*sizeof(int) );
    if( feld2 )
        exit( EXIT_FAILURE );
        /* ... */
}
```

Nur zur Sicherheit: Bei Strukturen (`struct`) wird nicht -wie bei Feldern- nur die Anfangsadresse übergeben, sondern ebenso wie bei einfachen Variablen eine vollständige Kopie (*call by value*), auch wenn in der Struktur möglicherweise wiederum Felder enthalten sein können!

In diesem Programm werden zwei Felder angelegt: `feld1` ist als Variable vom Typ *„Feld mit 10 int“* definiert; `feld1` dagegen als *„Zeiger auf int“* und erhält den Platz für das eigentliche Feld erst durch einen Aufruf von `malloc()` (siehe auch **8.5 Freier Speicher** auf Seite 48).

[17] Ausnahmen: Bitfelder und Registervariablen

In beiden Fällen kann man die einzelnen Feldelemente sowohl als `feld1[0]` (bzw. `feld2[0]`) bis `feld...[9]` ansprechen, als auch mit `*(feld1)` bzw. `*(feld2)` bis `*(feld1+9)` bzw. `*(feld2+9)`.

Ebenfalls gleichwertig sind die beiden Formen bei der Übergabe an eine Funktion. Sowohl der Ausdruck `feld1` als auch `feld2` liefert in einer Parameterliste oder Zuweisung die Adresse des jeweils ersten Feldelementes.

Nun zu den Unterschieden:

- Der Ausdruck `feld2` hat den Typ `int *`. `feld1` dagegen hat den Typ `int const *`, also ein *„Zeiger auf int, der nicht verändert werden darf“*. Daraus folgt, daß beispielsweise der Ausdruck `feld2++` zulässig ist[18], `feld1++` dagegen nicht.

- Obwohl `feld1` und `feld2` beide die Adresse des ersten Feldelementes liefern, sind `&feld1` und `&feld2` grundverschieden!

  `&feld1` liefert die Adresse des Objektes `feld1` (also das Objekt mit den Elementen `feld1[0]` bis `feld1[9]`). Das ist natürlich gleichzeitig die Adresse des ersten Feldelements, also `&feld1[0]` mit dem Typ `int const *`!

  `&feld2` liefert hingegen die Adresse der vereinbarten Variable, also die Adresse eines Zeigers, und hat damit erstens einen anderen Typ (nämlich `int const **`) und einen anderen Wert als `&feld2[0]`!

  Dazu ein kleines Testprogramm, das entsprechende Felder vereinbart und die Adressen ausgibt:

```
/* feaddr.c 31. 7.95 kw
 */

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

#define printp(p)  printf( "Der Ausdruck %10s hat den Wert 0x%#p\n", \
                           #p, (p)                                    \
                       )

void main()
{
  int
    feld1[10],
    *feld2;

  feld2 = malloc( 10*sizeof(int) );

  printp( feld1 );
  printp( &feld1[0] );
  printp( &feld1 );

  printp( feld2 );
  printp( &feld2[0] );
  printp( &feld2 );
}
```

[18] Nach dem einmaligen (!) Ausdruck `feld2++` wären die Feldelemente als `feld2[-1]` bis `feld2[8]` erreichbar! Nach einem zweiten `feld2++` besteht das Feld aus den Elementen `feld2[-2]` bis `feld2[7]`.

Die Ausgabe des Programms lautet in einem Beispiel:

```
Der Ausdruck      feld1 hat den Wert 0x0xbffff6b4
Der Ausdruck  &feld1[0] hat den Wert 0x0xbffff6b4
Der Ausdruck     &feld1 hat den Wert 0x0xbffff6b4
Der Ausdruck      feld2 hat den Wert 0x0x8049770
Der Ausdruck  &feld2[0] hat den Wert 0x0x8049770
Der Ausdruck     &feld2 hat den Wert 0x0xbffff6b0
```

Die Adressen liegen bei anderen Testläufen natürlich wohl kaum an der gleichen Stelle, aber folgendes läßt sich festhalten: `feld1`, `&feld1[0]` und `&feld1` bezeichnen alle dieselbe Adresse.

`feld2` und `&feld2[0]` sind ebenfalls gleichwertig, aber `&feld2` hat einen anderen Wert! `&feld2` ist nämlich nicht die Adresse des Feldes, sondern des Zeigers darauf.

Felder mit `0` Elementen: Man glaubt ja nicht, was es alles gibt. Kann man auch ein Feld ohne Elemente (also mit der Länge `0`) vereinbaren?

Laut ANSI-Standard muß das nicht möglich sein; mit einigen Compilern geht es (GNU-C, Visual C++), mit anderen nicht (Borland).

Für sich genommen, macht es sicher auch keinen Sinn.

Eine mögliche Verwendung wäre eine `struct`, an deren Ende (gegebenenfalls nach weiteren Nutzdaten) ein solches leeres Feld vereinbart ist.

Wenn man sich jetzt darauf verläßt, daß die Elemente einer `struct` in der Reihenfolge im Speicher liegen, in der sie bei der Deklaration angegeben sind (alle mir bekannten Compiler machen das tatsächlich), dann kann über das (für den Compiler eigentlich leere) Feld auf den Speicher direkt hinter der `struct` zugreifen; schon das nullte Feldelement liegt hinter der `struct`.

Das ist natürlich kriminell, solange man nicht dafür sorgt, daß einem der Speicher hinter der `struct` auch gehört.

Deshalb ist es sinnlos, von einer solchen `struct` direkt eine Variable zu vereinbaren; man hat ja keine Kontrolle darüber, was dahinter kommt.

Aber wenn man den Speicher für eine solche `struct` mit `malloc()` beschafft, steht es einem frei, als zu allokierende Größe etwas mehr anzugeben, und dementsprechend viele Elemente des Feldes verwenden zu können. Man muß natürlich wissen, wieviele Elemente allokiert wurden, um den Extraplatz nicht zu überschreiten. Dafür wird in solchen Fällen üblicherweise vor dem Feld noch eine ganzzahlige Variable (`size_t` oder ähnliches) vereinbart, in der die entsprechende Anzahl hinterlegt ist:

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

/* Datentyp, der ein mehr oder weniger langes Feld enthält:
 */
typedef struct
{
  /* hier vielleicht noch mehr Nutzdaten... */

  /* zusätzliches Feld; wieviele Elemente ist noch nicht bekannt; */
  /* das Feld muß ganz am Ende stehen!                            */
```

```
  size_t     anzahl;    /* Anzahl der Feldelemente                */
  int        feld[0];   /* Feldelemente                           */
}
MehrOderWeniger;

/* FuelleFeld() kann auf die Elemente des Feldes zugreifen, wenn der
 * Compiler mitspielt:
 * (1) muß der Compiler akzeptieren, daß in der obigen struct
 *     0 Elemente stehen
 * (2) muß der Compiler das leere Feld auch wirklich an das
 *     Ende der struct legen (meines Wissens machen das alle)
 */
void FuelleFeld( MehrOderWeniger *zuFuellen_p, int wert )
{
  size_t    i;
  for( i=0; i<zuFuellen_p->anzahl; i++ )
  {
    zuFuellen_p->feld[i] = wert;
  }
}

int main( int nargs, char **args )
{
  /* zwei Felder allokieren, und die Anzahl der zusätzlichen Elemente
   * eintragen:
   */
  MehrOderWeniger *p_mehr
    = (MehrOderWeniger*)malloc( sizeof(MehrOderWeniger) + 100*sizeof(int) );
  p_mehr->anzahl = 100;

  MehrOderWeniger *p_weniger
    = (MehrOderWeniger*)malloc( sizeof(MehrOderWeniger) + 5*sizeof(int) );
  p_weniger->anzahl = 5;

  /* Felder initialisieren:
   */
  FuelleFeld( p_mehr, 123 );
  FuelleFeld( p_weniger, 456 );

  return 0;
} // main( int nargs, char **args )
```

Beim Allokieren des Speichers wird angegeben, wieviele Elemente im Feld enthalten sein sollen, und diese Anzahl wird in der `struct` vermerkt.

Vereinbart man eine Variable eines solchen Typs direkt, hat man keine (legale) Möglichkeit, Platz für Feldelemente zu reservieren. Deshalb ist eine solche Spielerei auf frei allokierten Speicher beschränkt.

In Anbetracht der Tatsache, daß ein solches Programm kaum noch als portabel bezeichnet werden kann, sollte man nach Möglichkeit die Finger davon lassen. Allerdings ist es halt manchmal doch recht verlockend. . . ; beispielsweise gibt es in der Win32-API (Schnittstelle zu Windows) mehrere solcher `struct`s.

8.8.2 Strings

In Feldern vom Typ `char` kann man Strings, also Zeichenketten, speichern.

Strings sind in *C* normalerweise Folgen von Zeichen (jeweils vom Typ `char`, je ein Byte lang). Das Ende eines Strings ist mit einem Nullbyte, also dem Zeichen `'\0'`, markiert. Ein String mit 5 nutzbaren Zeichen benötigt demnach ein 6 Elemente großes `char`-Feld.

Schreibt man eine Zeichenkette mit doppelten Anführungszeichen in einem *C*-Programm (beispielsweise `"abc, die Katze liegt im Schnee"`), dann wird an die eigentlichen Zeichen immer automatisch eine Null angehängt; siehe dazu **8.1 Konstanten** auf Seite 31.

Diese Art von Zeichenketten heißt oft *null terminated string*; zur Arbeit damit existieren in der Standardbibliothek viele Funktionen:

`strcpy()` zum Kopieren, `strcat()` zum Aneinanderhängen von 2 Strings, `strchr()` um ein Zeichen in einem String zu finden, `strstr()` um eine Zeichenfolge in einem String zu finden, und andere.

Alle diese Funktionen verlassen sich darauf, daß die Zeichenketten mit dem Nullbyte abgeschlossen sind.

Wie in **8.8.1 Felder** auf Seite 61 beschrieben, wird bei der Übergabe eines Feldes an eine Funktion tatsächlich stattdessen ein Zeiger auf das erste Element des Feldes übergeben.

`strcpy()` und `strcat()` erhalten zwei Zeiger vom Typ `char *`, und liefern selbst einen Zeiger vom gleichen Typ zurück. Die Vereinbarung sieht also so aus:

```
char *strcpy( char *wohin, char *woher );
char *strcat( char *woran, char *woher );
```

und ist in der Datei string.h definiert.

Beispiel:

```
/* stgut.c 31. 7.95 kw
 */

/* Die Praeprozessordirektive #include dient dazu,
 * die Datei string.h hier einzufuegen.
 * In string.h sind unter anderem die Funktionen strcpy()
 * und strcat() vereinbart.
 * Stattdessen haette man die Funktionen auch hier direkt
 * vereinbaren koennen:
 */

#include <string.h>

/* Dann vereinbaren wir etwas Platz fuer einen String:
 */

char     s[20];

main()
{
```

```
  /* In der naechsten Zeile wird eine Stringkonstante
   * mit den 5 Zeichen 'S', 'O', 'S', ' ' und '\0'
   * erzeugt. Dann werden an die Funktion strcpy zwei
   * Zeiger vom Typ char * uebergeben, naemlich ein
   * Zeiger auf das erste Byte in s und ein Zeiger auf
   * das erste Byte in der Stringkonstanten, also auf
   * das 'S'.
   * Die Funktion kopiert alle 5 Zeichen aus der
   * Konstanten einschliesslich der abschliessenden '0'
   * an den Anfang des Feldes s. In s stehen danach
   * die 5 Zeichen und weitere 15, die keiner so genau
   * kennt.
   * Der Rueckgabewert von strcpy ist wieder ein Zeiger
   * auf das erste Byte der Kopie, also auf s[0]. Der
   * Rueckgabewert wird hier aber nicht verwendet.
     */
  strcpy( s, "SOS " );

  /* Nun werden analog zur letzten Anweisung die 5
     * Zeichen 'D', 'a', 's', ' ' und '\0' an den
     * Anfang von s kopiert. Die ersten 5 Zeichen, die
     * schon dort stehen, werden damit ueberschrieben und
     * sind verloren.
     */
  strcpy( s, "Das " );

  /* Mit strcat() werden an den bisherigen String in s
     * die Zeichen "ist g" und eine abschliessende '\0'
     * angehaengt, so dass sich ein zusammenhaengender
     * String "Das ist g" mit einer abschliessenden '\0'
     * ergibt.
     * Die bisherige abschliessende '\0'in c[4] wird
     * dabei von dem 'i' aus "ist g" ueberschrieben.
     */
  strcat( s, "ist g" );

  /* An den bisherigen String "Das ist g" werden analog
     * noch die Zeichen 'u', 't', '!', das Zeilen-
     * vorschubzeichen '\n' und eine abschliessende '\0'
     * angehaengt.
     */
  strcat( s, "ut!\n" );

  /* s enthaelt nun den String "Das ist gut!\n" mit der
     * beruehmten '\0' am Ende sowie 6 weitere,
     * ungenutzte Zeichen, deren Inhalt keiner weiss.
     * Der String wird mit printf() jetzt bis zur '\0'
     * ausgegeben; die uebrigen 6 Zeichen stoeren dabei
     * nicht, weil printf() bei der ersten '\0' im
     * auszugebenden String abbricht.
     */
  printf( s );
}
```

Achtung! In *C* werden standardmäßig keine Feldgrenzenüberprüfungen vorgenommen.

Deshalb werden auch Stringoperationen (z.B. mit `strcpy()`) ohne jede Kontrolle der Länge durchgeführt!

8.9 Initialisierung

Beim Definieren von Variablen kann man sie in *C* gleich initialisieren. Dies geht einfach, indem man nach der Definition mit `=` einen Wert angibt:

```
int
    i = 1;
double
    d1 = 25.8,
    d2;
```

vereinbart drei Variablen und gibt `i` den Anfangswert `1` und `d1` den Anfangswert `25.8`; `d2` wird ebenfalls vereinbart, aber nicht initialisiert.

Strukturen und Felder initialisiert man, indem man für jede Struktur oder Teilstruktur die Werte in geschweiften Klammern ({...}) zusammenfaßt. Beginnt man eine Initialisierung, gibt aber nicht alle Werte an, dann werden die restlichen Elemente mit Nullen initialisiert (ein guter Compiler sollte hier aber eine Warnung ausgeben).

```
struct
{
    int     a;
    double  d;
}
v = { 25, 3.1415 };
```

vereinbart eine Variable `v` mit den Elementen `a` und `d`. `v.a` erhält den Anfangswert 25; `v.d` den Wert `3.1415`.

```
int    feld[5] = { 1, 3, 5 };
```

vereinbart ein Feld mit den 5 Elementen `feld[0]` bis `feld[4]`. `feld[0]` erhält den Wert `1`, `feld[1]` den Wert 3 und `feld[2]` den Wert `5`. Für `feld[3]` und `feld[4]` ist kein Wert mehr angegeben; diese Elemente erhalten beide den Wert `0`.

Gibt man alle Werte an, dann kann man dafür die Angabe der Feldlänge weglassen (muß aber nicht).

```
int feld[] = { 1, 3, 5, 0, 0 };
```

vereinbart ebenfalls ein Feld mit 5 Elementen; der Compiler kann die Feldlänge aus der Anzahl der Initialisierungen ablesen.

Für Felder vom Typ `char[]` gibt es eine abkürzende Schreibweise.

So kann man:

```
char cfeld[5] = { 'a', 'b', 'c', 'd', 'e' };
```

abkürzen durch[19]:

```
char cfeld[5] = { "abcde" };
```

Die fünf angegebenen Werte sind tatsächlich nur Anfangswerte und dürfen während des Programmlaufs geändert werden, etwa so:

```
    cfeld[3] = 'f';
```

Dies darf man nicht verwechseln mit der Initialisierung eines Zeigers:

```
char  *p = "abcde";
```

Damit vereinbart man einen Zeiger `p`, der anfangs auf die Stringkonstante `¨abcde¨` (6 Elemente inklusive Null) zeigt. Den Inhalt der Konstante, also die Zeichen, darf man nicht ändern (kann es aber trotzdem auf einigen Systemen). Ändern darf man dagegen den Inhalt des Zeigers selbst; dieser darf also irgendwann zur Laufzeit auch mal auf andere Zeichen an anderen Stellen im Speicher zeigen.

Felder oder Strukturen, die aus Feldern oder Strukturen bestehen, werden entsprechend ihrer Reihenfolge im Speicher mit geschachtelten geschweiften Klammern {...} initialisiert,

Bei der Initialisierung von automatischen Variablen (siehe **8.6 Attributangaben** auf Seite 50) darf man auch einen Ausdruck angeben, der erst zur Laufzeit ausgewertet wird, also beispielsweise ein Rechenausdruck, der Funktionsparameter verwendet, oder ein Funktionsaufruf:

```
/* ingibini.c 31. 7.95 kw
 */

int gibi(void)
{
  return 5;
}

int main( int nargs, char **args )
{
  int n = nargs*2;
  int i = gibi();

  printf( "n=%d, i=%d\n", n, i );
}
```

[19] Achtung! In diesem Beispiel wird ausdrücklich ein Feld mit 5 Elementen definiert und mit 5 Zeichen initialisiert. Dadurch wird keine abschließende Null angehängt!

Statische Variablen (also auch globale) darf man nur mit konstanten Ausdrücken initialisieren.

Initialisieren sollte man natürlich nur mit Werten, die auch vom Typ her mit den Variablen verträglich sind. Beispielsweise kann man `int`-Variablen mit ganzen Zahlen initialisieren und Zeiger auf Funktionen mit den Adressen von Funktionen; aber nicht umgekehrt.

Globale Variablen, also außerhalb von Funktionen definierte, werden automatisch mit Nullen initialisiert, sofern man nichts anderes angibt[20].

9 Speichermodell eines Prozesses

Um das Verhalten von *C*-Programmen besser zu verstehen, ist es hilfreich, eine (vereinfachte) Vorstellung von *Programmen* und *Prozessen* zu haben.

Ein *Programm* (oder ein Teil eines Programmes) kann in mehreren Formen auftreten, unter anderem:

- als formfreie oder formalisierte Handlungsanweisung (Algorithmus)
- als Quelltext in einer bestimmten Programmiersprache wie *C*
- als Objektcode (das, was ein Compiler aus dem Quelltext produziert)
- als ausführbares Programm in Form einer Datei (das Ergebnis des Linkvorgangs)
- nach dem Starten (*Laden*) der Programmdatei als laufendes Programm im Speicher

Die letztere Form, also das Programm während der Ausführung, heißt üblicherweise *Prozeß*.

Je nach verwendetem Rechnersystem können solche Prozesse unterschiedlich dargestellt werden, aber viele Gemeinsamkeiten lassen sich erkennen, wenn man die Betrachtung weit genug vereinfacht. Im Folgenden beschränke ich mich auf wenige Punkte, die für das Verständnis von *C*-Programmen wesentlich sind; mehr Einzelheiten sind in allgemeineren Einführungen in die Informatik zu finden.

Ein Prozeß kann nur laufen, wenn eine entsprechende Systemumgebung vorhanden ist:

- *Arbeitsspeicher* (*random access memory*, RAM) ist ein Speicher für Daten aller Art, die während der Ausführung eines Prozesses vorgehalten werden müssen, also für den auszuführenden Programmcode, die meisten Variablen, Zwischenergebnisse und ähnliches.

  Der Arbeitsspeicher, den ein Prozeß sieht, besteht einfach aus einer langen Folge von Werten, die jeweils über eine *Adresse* angesprochen werden können. Eine solche Adresse ist nichts weiter als eine ganze Zahl zwischen 0 und einer systemabhängigen Obergrenze (typischerweise einige Megabyte bis einige Gigabyte). Eine Adresse ist in *C* als Wert eines Zeigers darstellbar und kann für alle dem Compiler bekannten Objekte mit Adreßoperator (`&`) beschafft werden.

[20] Eine Ausnahme davon ist aber der Kernel von Unix-ähnlichen Betriebssystemen! Hier werden keine Variablen automatisch initialisiert.

Tatsächlich wird bei üblichen Systemen zwischen dem *realen* und dem *virtuellen* Speicher unterschieden. Auf den Unterschied wird hier nicht eingegangen; im Zweifelsfall geht es im Folgenden nur um den virtuellen Speicher.

- ein *Prozessor* (*central processing unit*, CPU), der die Befehle des Programms (die in Maschinensprache vorliegen müssen) in der richtigen Reihenfolge ausführt.

  Ein Prozessor hat neben verschiedenen funktionalen Einheiten (Rechenwerke, Steuerwerk etc.) einen Satz von *Registern*. Das sind sehr schnelle Speicher für Rechenoperanden, Ergebnisse usw.; ihre Anzahl ist begrenzt und sehr klein im Vergleich zum gesamten Prozeß (in der Regel nur für wenige oder maximal einige Dutzend Werte).

  Typische Prozessoren haben einige allgemein verwendbare Register für ganze Zahlen, meist auch welche für Gleitkommazahlen, sowie Adreßregister, die zum Ansprechen von Daten im Arbeitsspeicher verwendet werden.

  Einige Register haben eine besondere Bedeutung:

  - der *program counter* (meist *PC* genannt) enthält die Adresse des nächsten auszuführenden Maschinenbefehls.

    Während ein Prozeß läuft, muß sein gesamter Programmcode im Arbeitsspeicher erreichbar sein.

    Bei jedem Laden eines Befehls aus dem Speicher wird der PC um die Länge des gelesenen Befehls automatisch weitergesetzt.

    Nach der Ausführung eines Befehls wird der nächste Befehl anhand des PC aus dem Speicher gelesen.

    Programmsprünge werden durchgeführt, indem dieses Register mit einer anderen Adresse überschrieben wird.

  - Der *stack pointer* (SP, bei Intelprozessoren auch *esp* genannt) und der *frame pointer* (FP, bei Intel *bp* („base pointer“) oder *ebp* genannt) sind zwei eng verwandte Adreßregister, die in den weiter unten beschriebenen *Stack* zeigen.

- sowie viele weitere Dinge, die hier nicht wichtig sind (Hardware, um Daten einzulesen oder auszugeben, ...)

Der Arbeitsspeicher eines Prozesses ist nun zumindest in folgende Bereiche, sogenannte *Segmente* unterteilt:

- Das *Codesegment* (oft auch *Textsegment* genannt) enthält nach dem Laden des Programms den gesamten auszuführenden Programmcode (in Maschinensprache); in diesen Bereich zeigt hoffentlich der program counter.

- Das Datensegment (*data segment*) enthält alle Daten, die während des gesamten Programmlaufs genau einmal existieren (alle globalen Variablen sowie die lokalen statischen), soweit sie bei Programmstart initialisiert werden müssen.

- Das *block storage segment* (BSS) enthält analog alle globalen und lokalen statischen Variablen, die nicht initialisiert werden müssen. Die Unterscheidung zwischen initialisierten und nicht initialisierten Variablen wird üblicherweise getroffen, weil initialisierte Daten durch ihre Initialisierungswerte Platz in der Programmdatei belegen, den man sich für nicht initialisierte Variablen sparen kann (es genügt dem Betriebssystem, beim Laden die Gesamtgröße aller BSS-Daten zu kennen).

- Der Stack (*stack segment*) nimmt Daten auf, die dynamisch zur Laufzeit in einer geordneten Reihenfolge entstehen und genau in der umgekehrten Reihenfolge nicht mehr benötigt werden.

  Solche Daten sind:

  - alle Argumente an aufgerufene Funktionen
  - deren automatische Variablen (alle lokalen Variablen, die nicht als `static` deklariert sind)
  - Rücksprungadressen (die Stelle im Programm, an der nach dem Ende einer Funktion fortgefahren werden soll)
  - Registerinhalte, die für die Dauer eines Funktionsaufrufs in Sicherheit gebracht werden sollen
  - teilweise Platz für Rückgabewerte von Funktionen (in *C* typischerweise für die Rückgabe von nicht elementaren Datentypen, also Feldern oder Strukturen)
  - kurzfristig benötigter Platz für Zwischenergebnisse, beispielsweise bei arithmetischen Ausdrücken

  Dazu wird einfach bei Programmstart ein ausreichend großer Bereich im Speicher reserviert (eben der Stack). Der Stack besteht immer aus zwei Teilen: ein zusammenhängender Bereich, der bereits belegt ist (üblicherweise der obere Teil), sowie der gesamte Rest, der noch frei ist.

  Der Stackpointer zeigt nun zu jedem Zeitpunkt genau auf die Grenze zwischen dem freien und dem belegten Teil.

  Um weitere Werte im Stack abzulegen, wird der Stackpointer um die Länge des Wertes erniedrigt (falls wie üblich der belegte Bereich über dem noch freien liegt), und der zu speichernde Wert an die Stelle kopiert, auf die der SP zeigt.

  Der unterste Wert auf dem Stack (und nur dieser) kann freigegeben werden, indem der SP um die richtige Länge erhöht wird.

  Der Aufruf einer *C*-Funktion sieht nun so aus, daß die zu übergebenden Parameter (von rechts her) nacheinander auf dem Stack abgelegt werden, dann werden zu rettende Register (zumindest der program counter und der gleich noch beschriebene frame pointer) abgelegt, und letztlich wird der SP noch um soviele Byte erniedrigt, wie alle automatischen Variablen Platz benötigen.

  Wenn eine laufende Funktion `a()` eine weitere (`b()`) aufruft, wiederholt sich dieses Spiel: die an `b()` zu übergebenden Parameter, zu rettende Register und die von `b()` benötigten Variablen werden unterhalb des bereits belegten Stacks angelegt. Dadurch besteht zu jedem Zeitpunkt der belegte Stack aus eine Folge sogenannter *stack frames* aller gerade aktiven Funktionen. Ein *stack frame* einer Funktion enthält also die Parameter an die Funktion, gerettete Register, und lokale automatische Variablen.

  Wenn eine aufgerufene Funktion mit ihrem Programmcode beginnt, kann sie nun auf alle ihre automatischen Variablen ebenso wie auf ihre Parameter zugreifen, indem sie zum Wert des SP einen bestimmten Offset addiert (der dem Compiler bereits bekannt ist, weil er die Größen aller Datentypen und damit alle Positionen im stack frame kennt; dieser Mechanismus heißt *SP-relative Adressierung*).

  Während die Funktion aber läuft, wird sie weitere Werte auf dem Stack ablegen (zum Beispiel für Unterprogrammaufrufe) und sie wieder entfernen. Dadurch ändert sich der Wert des SP laufend, und auf ein und dieselbe Variable müßte ständig mit anderen Offsets zugegriffen werden. Um diesen Aufwand zu vermeiden, wird beim Start jeder Funktion der gerade aktuelle Wert des SP in das *frame pointer*-Register (FP) kopiert; dieses Register bleibt dann während der Abarbeitung der Funktion unverändert, und es kann anstatt SP-relativ über den FP auf automatische Variablen und Parameter zugegriffen werden, während sich der SP durchaus ändern kann.

Wird nun eine weitere Funktion aufgerufen, dann muß logischerweise der Inhalt des FP auf dem Stack gerettet werden, damit nach der Rückkehr wieder die lokalen Variablen korrekt adressiert werden, auch wenn die aufgerufene Funktion den FP zur Adressierung ihrer Variablen benutzt hat.

- Der gesamte, nicht von den genannten Segmenten belegte Speicher, steht zur freien Verfügung des Programms, und heißt meistens *Heap*.

  Dieser Heap wird in beliebiger Reihenfolge stückweise reserviert und wieder freigegeben, in *C* üblicherweise durch `malloc()` und `free()`.

Alle diese Zusammenhänge sind im Abbildung 4 skizziert.

Der dargestellte Quelltext definiert einige Objekte im Speicher (die Einfärbung der einzelnen Elemente entspricht den zugehörigen Speicherbereichen):

- zwei globale Variablen: die initialisierte Variable `global1` wird im Datensegment liegen, während die nicht initialisierte Variable `global2` im BSS untergebracht wird.
- `up()` bekommt zwei Parameter `a` und `b`, die beim Aufruf der Funktion auf dem Stack angelegt werden
- In `up()` sind zwei lokale Variablen definiert: `summe` ist eine automatische Variable, die auf dem Stack angelegt wird, während `zaehler` als `static` deklariert ist und bei Programmstart initialisiert wird; deshalb liegt sie Datensegment.
- `main()` wird ebenfalls mit zwei Argumenten aufgerufen (`nargs` und `args`), die sich auf dem Stack befinden.
- `i1` und `p` sind lokale automatische Variablen im Stacksegment.
- Von `malloc()` wird Speicher im Heap reserviert: die Objekte `p[0]` bis `p[9]` sind demnach im Heap (aber `p`, also die Zeigervariable, die sich die Heapadresse merkt, liegt im Stack!).

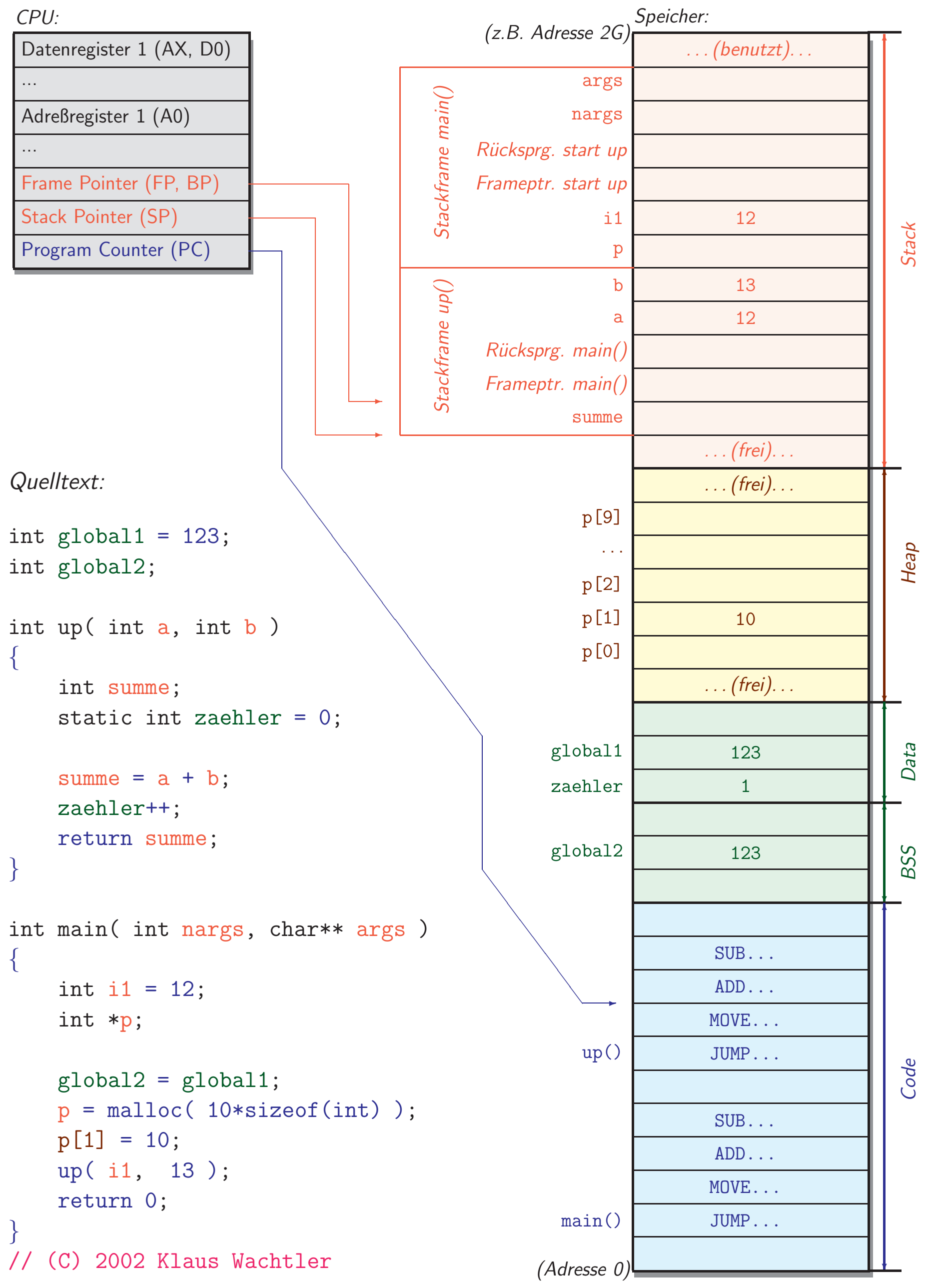


Abbildung 4: Typische Speichersegmente eines Prozesses

10 Funktionen

Programmcode wird in *C* immer in Form von Funktionen definiert; eine formale Unterscheidung zwischen Hauptprogramm, Unterprogrammen und Funktionen wie in anderen Sprachen gibt es nicht (*Pascal*: `PROCEDURE/FUNCTION`, *FORTRAN*: `SUBROUTINE/FUNCTION`). Dafür müssen Funktionen nicht unbedingt einen Wert zurückgeben. Geben sie einen Wert zurück, muß der Aufrufer den Wert nicht verwenden.

In *FORTRAN* etwa ist folgendes streng verboten:

```
        ...
        INTEGER FUNCTION summe( a, b )
        INTEGER a, b
        summe = a + b
        END
        ...
        INTEGER   summe;
*         erlaubt:
        a = summe( 1, 2 );
*         verboten:
        CALL summe( 3, 4 );
        ...
```

Das Gegenstück in *C* ist aber vollkommen legal[21]:

```
/* ... */
int summe( int a, b )
{
    return a + b;
}
/* ... */
    a = summe( 1, 2 );  /*  erlaubt            */
    summe( 3, 4 );      /* auch erlaubt        */
/* ... */
```

Ein Hauptprogramm im Sinne von anderen Sprachen gibt es nicht; die Programmausführung beginnt immer bei einer Funktion namens `main()`.

Funktionen dürfen —im Gegensatz zu *Pascal* oder *Modula2*— nicht innerhalb einer anderen Funktion definiert werden. Alle Funktionen in einem *C*-Quelltext stehen hintereinander und nicht ineinander geschachtelt.

Funktionsdefinitionen bestehen aus einem Kopf, in dem die Deklaration erfolgt (gegebenenfalls Attribute, der Typ des Rückgabewerts, der Name der Funktion, ein paar runde Klammer (`(`...`)`), die bei Bedarf die Parameter deklarieren) und einem Block, in dem lokale Variablen und der Funktionsablauf definiert werden. Ein Block ist in *C* immer in geschweifte Klammern (`{`...`}`) eingeschlossen (das entspricht dem Paar `BEGIN`...`END` in *Pascal*).

Beispiel:

```
/* fupotenz.c 30. 7.95 kw
 */
```

[21] In diesem Beispiel aber natürlich unsinnig!

```
float potenz( float basis, int exponent )
{
  int i;
  float ergebnis;

  ergebnis = 1.0;
  for( i=0; i<exponent; i=i+1 )
  {
    ergebnis = ergebnis*basis;
  }
  return ergebnis;
}
```

Im Funktionskopf wird vereinbart, daß die Funktion namens `potenz()` einen Rückgabewert vom Typ `float` liefert (Gleitkommazahl in einfacher Genauigkeit) und dazu einen Parameter vom Typ `float` und einen vom Typ `int` erhält.

Im Funktionsrumpf (äußeres `{}`-Paar) werden erst die Variablen vereinbart (`i`, `ergebnis`) und dann der eigentliche Programmcode definiert.

Dieser wiederum besteht aus einer Zuweisung (`ergebnis = 1.0;`), einer `for`-Schleife und einer `return`-Anweisung.

In der `for`-Schleife wird zuerst `i` auf den Wert `0` gesetzt. Dann wird die Folge

- Laufbedingung `i<exponent` prüfen; gegebenenfalls die Schleife verlassen,
- Schleifenrumpf ausführen (`{ ergebnis = ...; }`),
- Reinitialisierung ausführen (`i = i+1`)

durchgeführt, bis die Schleife wegen Nichterfüllung der Laufbedingung verlassen wird.

Mit der `return`-Anweisung wird die Abarbeitung der Funktion beendet und der Wert zurückgeliefert, der hinter dem `return` steht. Ist als Rückgabewert einer Funktion `void` vereinbart, dann darf hinter der `return`-Anweisung kein Rückgabewert stehen!

Lokale Variablen von Funktionen (wie hier `i`) stehen nur während des Abarbeitens der aktuellen Funktion zur Verfügung und können beim nächsten Aufruf derselben Funktion in der Regel ganz andere Werte haben. Das kann man aber auch anders haben; siehe **8.6 Attributangaben** auf Seite 50.

Das eigentliche Hauptprogramm ist auch nur eine Funktion, die einen Wert vom Typ `int` liefert. In aller Regel sollte das der Wert `0` sein. Diese Funktion hat den Namen `main()` und spielt die Rolle des Hauptprogramms in anderen Sprachen.

Funktionsparameter müssen in *C* nicht unbedingt deklariert werden, es ist aber immer empfehlenswert, um wenigstens einige Fehlermöglichkeiten auszuschließen.

Werden Parameter nicht deklariert, bevor eine Funktion aufgerufen wird, dann schließt der Compiler aus den übergebenen Parametern auf den jeweiligen Typ, den die Funktion erwartet. `char`-Parameter werden dann zu `int` gewandelt und `float` zu `double`. Felder werden als Zeiger auf das erste Element übergeben. Alle anderen Parameter werden (als Kopie, *call by value*, siehe unten) unverändert an eine Funktion weitergereicht[22].

Die Deklaration einer Funktion könnte etwa so aussehen:

```
double ldurchi( long l, int i );
```

Das bedeutet, daß die Funktion `ldurchi()` zwei Parameter erhält, nämlich den ersten vom Typ `long`, und einen zweiten vom Typ `int`. Weiterhin besagt die Vereinbarung, daß die Funktion ein Resultat vom Typ `double` liefert.

Nach der obigen Deklaration könnte man die Funktion beispielsweise so aufrufen:

```
printf( "%f", ldurchi( 2, 3 ) );
```

Hätte man die Funktion nicht deklariert, dann hätte der Compiler keine reelle Chance, die richtigen Datentypen für den Aufruf zu erraten. In diesem Fall würde er annehmen, daß zwei Parameter vom Typ `int` übergeben werden sollen, weil ja solche Parameter verwendet werden, und als Rückgabewert den Standard `int` vermuten. Der Erfolg eines Aufrufs wäre dann sehr zweifelhaft.

C kennt eigentlich nicht die Unterscheidung zwischen *call by value* und *call by reference*. Letztere sind in *Pascal* die als `VAR` vereinbarten Funktionsparameter. In *FORTRAN* gibt es nur *call by reference*, das heißt daß Änderungen eines Parameters innerhalb eines Unterprogramms auch die entsprechende Variable beim aufrufenden Modul ändern.

In *C* gibt es formal nur Parameter, die mit dem *call by value* -Mechanismus übergeben werden.

Einzige Ausnahmen davon sind **Felder** (Seite 61) und **Funktionen** (Seite 78) (diese beiden werden generell als Adreßparameter (*call by reference*) übergeben).

Wird also eine Funktion mit Parametern aufgerufen, dann erhält sie in der Regel tatsächlich nur Kopien der Parameter. Ändert eine Funktion die Werte ihrer Parameter, dann wirkt sich dies nicht beim Aufrufer aus.

```
/* fu55.c 30. 7.95 kw
 *
 * Die Funktion f() erhoeht ihren Parameter und beendet
 * sich dann gleich wieder.
 */
void f( int wert )
{
  printf( "%d\n", wert ); /* Den Parameter ausgeben      */
  wert = wert + 1;        /* und erhoehen.               */
}

main()
{
  int     i;              /* Variable i vereinbaren      */

  i = 5;                  /* Den Wert 5 zuweisen         */
  f( i );                 /* Funktion f aufrufen         */
```

[22]Im älteren K&R-*C* wurde noch eine weitere automatische Umwandlung durchgeführt, und zwar wurden Strukturen übergeben als Zeiger auf die Struktur. Will man dies in *ANSI-C* so haben, dann muß man es explizit mit dem Adreßoperator ausdrücken.

```
  printf( "%d\n", i );    /* Den Wert von i ausgeben     */
}
```

gibt zweimal die Zahl 5 aus, obwohl die Funktion `f()` ihren Parameter um eins erhöht:

```
5
5
```

`f()` hat tatsächlich keinen Zugriff auf die Variable `i` aus `main()`, sondern hat nur ein Kopie davon zur Verfügung.

C wäre aber nicht *C*, wenn man dieses Verhalten nicht umgehen könnte.

Um *call by reference* zu erzwingen, kann man an die Funktion statt der Variablen deren Adresse übergeben und innerhalb der Funktion statt des `int`-Parameters einen solchen vom Typ `int *` verwenden, über den man auf die Variable zugreift. Dann wirken sich Änderungen an der Variable auch beim Aufrufer aus, da über den Zeiger ja die Variable selbst manipuliert werden kann:

```
/* fu56.c 30. 7.95 kw
 */

void f( int *zeig )          /* Statt des Werts jetzt einen
                              * Zeiger darauf.
                              */
{
  printf( "%d\n", *zeig );   /* An den eigentlichen Wert    */
  (*zeig)++;                 /* kommen wir jetzt mit *zeig  */
                             /* heran (das, worauf zeig     */
                             /* zeigt).                     */
}

main()
{
  int     i;
  i = 5;
  f( &i );                   /* Nicht den Wert uebergeben,
                              * sondern seine Adresse (&i)
                              */
  printf( "%d\n", i );
}
```

erzeugt die Ausgabe:

```
5
6
```

Die Variable `i` des Hauptprogramms wurde also tatsächlich manipuliert.

Rückgabewert von Funktionen: Soll eine Funktion einen Wert zurückgeben, dann ist sie natürlich erstmal mit dem entsprechenden Funktionskopf zu vereinbaren. Ist bei der Deklaration oder Definition vor dem Funktionsnamen kein Datentyp angegeben, dann wird vom Compiler der Typ `int` für den Rückgabewert angenommen.

Will man dies verhindern, weil die Funktion gar keinen Wert zurückgeben soll, dann wird sie mit dem Typ `void` vereinbart. Hinter den `return`-Anweisungen der Funktion darf dann kein Rückgabewert angegeben sein.

Soll die Funktion einen Wert liefern, dann muß man hinter den `return`-Anweisungen einen Ausdruck angeben, der das gewünschte Ergebnis liefert.

Beispiel:

```
double pi()
{
  return 3.14159265;
}
```

11 Operatoren und Ausdrücke

Die Operatoren in *C* sind:

Operator	*Zusammenfassung*	*Bedeutung*
`() []` `-> .`	von links	Klammeroperator, Feldzugriff Elementauswahl für `struct/union`

`! ~`
`++ -- + -`
`* &`
`(...) sizeof`
	von rechts	log. und bin. Verneinung
		Inkrement, Dekrement, Vorzeichen
		Zeigerverweis, Adreßoperator
		cast, Speichergröße

`* / %`	von links	Multiplikation, Division, modulo
`+ -`	von links	Addition, Subtraktion
`<< >>`	von links	links, rechts Schieben
`< <= > >=`	von links	numerischer Vergleich
`== !=`	von links	Gleichheit, Ungleichheit
`&`	von links	binäres Und (AND)
`^`	von links	binäres Exklusiv-Oder (XOR)
`|`	von links	binäres Oder (OR)
`&&`	von links	logisches Und (AND)
`||`	von links	logisches Oder (OR)
`?:`	von rechts	Auswahloperator

`= += -= *= /=`
`%= &= ^= |=`
`<<= >>=`
	von rechts	Zuweisungen

`,`	von links	Kommaoperator

Dabei haben `()`, `[]`, `->` und `.` aus der obersten Gruppe die höchste Priorität vor den Operatoren aus der zweiten Gruppe usw.; innerhalb jeder Gruppe haben alle Operatoren gleichen Rang.

Die Auswertungsreihenfolge läßt sich wie gewohnt mit Klammerung `( ...)` übersteuern.

Einige Operatoren tauchen mehrmals auf (z.B. `+`). Ihre Bedeutung richtet sich dann danach, ob sie sich im gegebenen Zusammenhang auf einen (monadischer Operator) oder auf zwei (diadischer Operator) Operanden beziehen.

Die Operatoren bedeuten im einzelnen:

- Der Feldelementzugriff mit eckigen Klammern `[ ...]` wird in **8.8.1 Felder** auf Seite 61 beschrieben, dort sind auch Beispiele zu finden.

  Der Operand ist nur zulässig für Felder und Zeiger; der Index muß ein ganzzahliger Ausdruck sein.

- Der Punkt `.` als Auswahloperator für Strukturen und `union`s ist in **8.7 struct und union** auf Seite 53 erklärt.

  Der linke Operand muß eine `struct` oder `union` sein; der rechte ein zugehöriges Element.

  Oft kommt es vor, daß ein Zeiger, z.B. `p`, auf eine Struktur zeigt. Angenommen, die Struktur enthält ein Element mit dem Namen `el`, dann müßte man zum Zugriff auf dieses Element schreiben:

  ```
  (*p).el
  ```

Hier ist die Klammerung nötig, da der Punktoperator höhere Priorität hat als der Verweis mit `*`, obwohl ja erst der Verweis und dann die Auswahl durchgeführt werden muß.

Deshalb gibt es als abkürzende Schreibweise den Operator `->`. Das Beispiel kann man also auch so schreiben:

```
p->el
```

Ansonsten sind beide Formen gleichwertig.

- `!` verneint logische Ausdrücke; `~` verneint binäre Ausdrücke bitweise.

  Letzterer ist nur zulässig für ganzzahlige Operatoren.

- `++` und `--` erhöhen beziehungsweise erniedrigen ihren Operanden um den Wert `1` (außer für Zeiger: hier wird die Größe des verwiesenen Objekts addiert oder subtrahiert; siehe **8.4.4 Zeiger** auf Seite 41). Sie dürfen vor oder nach dem Operanden stehen; dementsprechend wird der umgebende Ausdruck mit dem schon manipulierten Wert des Operanden oder noch mit dem bisherigen bewertet.

  ```
  i = 5;
  j = ++i;
  ```

  weist an `i` erst den Wert `5` zu und erhöht `i` dann auf `6`; an `j` wird der schon erhöhte Wert, also `6` zugewiesen.

  ```
  i = 5;
  j = i++;
  ```

  dagegen weist an `j` noch den alten Wert, also `5`, zu und erhöht `i` ebenfalls auf `6`.

  Zulässig für alle veränderbaren Grunddatentypen (also nicht für Feldnamen, nicht für Strukturen, Felder etc.).

- `+` und `-` (monadisch) als Vorzeichen dürften gemeinhin bekannt sein.

- `*` als Verweisoperator (monadisch) liefert das, worauf ein Zeigerwert verweist.

  Hat man die Vereinbarung

  ```
  double *p;
  ```

  dann meint `p` den Wert des Zeigers, also eine Adresse. `*p` dagegen liefert den `double`-Wert, auf den `p` zeigt.

  Natürlich nur zulässig für Zeiger.

- `&` (monadisch) ist die Umkehrung davon: geliefert wird die Adresse einer Variablen anstatt ihres Wertes. Mit der Vereinbarung

  ```
  double  d;
  ```

  meint `d` den Inhalt von `d`, während `&d` die Speicherstelle liefert, an der `d` zur Laufzeit liegt.

  Zulässig für alle Objekte im Speicher, wie Variablen, Feldelemente, Funktionen, über Zeiger adressierte Objekte.

  Nicht zulässig für Registervariablen und Bitfelder.

- `( ... )` (wenn in dem Klammerpaar ein Typ angegeben ist) ist der sogenannte *cast*-Operator. Mit ihm läßt sich eine Typanpassung erzwingen. Der *cast*-Operator wird vor den umzuwandelnden Ausdruck geschrieben. In den Klammern wird der gewünschte Typ eingetragen.

  ```
  int i;
  double d;
  i = 1;
  d = (double)i;
  ```

  verwandelt den Ausdruck `i`, der den Typ `int` liefert, in einen `double`-Wert und weist diesen an `d` zu.

  In diesem Fall hätte man sich die explizite Umwandlung mit dem *cast* `(double)` auch sparen können, weil bei der Zuweisung

  ```
  d = i;
  ```

  genau die gleiche Umwandlung automatisch vorgenommen wird.

  Ein ge*cast*eter Ausdruck ist nie ein Objekt, dem etwas zugewiesen werden kann („*lvalue*“).

- `sizeof` ist ein Operator, der die Größe seines Operanden im Speicher liefert (gemessen in Bytes). Der Operand darf ein Objekt einfachen oder zusammengesetzten Typs im Speicher sein oder ein Typname. In letzterem Fall muß der Typname geklammert sein (z.B. `sizeof(int)`).

- `*` (diadisch) und `/` führen eine Multiplikation beziehungsweise Division aus.

- `%` berechnet für ganzzahlige Operanden (*linker Operand* modulo *rechter Operand*).

  Für Gleitkommaoperanden gibt es die Funktion `fmod()`; siehe `fmod()` (S. 192).

- `+` und `-` (diadisch) führen Addition und Subtraktion durch.

  Zulässig für zwei arithmetische Operanden oder für einen ganzzahligen Operanden und einen Zeiger (oder Feld).

  Im letzten Fall wird der ganzzahlige Operand implizit mit der Speichergrösse des Objekts multipliziert, auf das der Zeiger verweist.

  Dabei ist der Ausdruck `*(`*Zeiger*+*ganze_Zahl*`)` identisch mit *Zeiger*`[`*ganze_Zahl*`]`, aber (wegen der Kommutativität der Addition) auch mit `*(`*ganze_Zahl*+*Zeiger*`)`. Deshalb kann man in *C* ungestraft den Ausdruck *Zeiger*`[`*ganze_Zahl*`]` ersetzen durch den gleichwertigen Ausdruck *ganze_Zahl*`[`*Zeiger*`]`.

- `<<` und `>>` verschieben den Wert links vom Operator um so viele Stellen nach links oder rechts, wie rechts des Operators angegeben ist. Dabei fallen auf der einen Seite gültige Binärstellen weg, auf der anderen werden Nullen nachgezogen (arithmetisches Shiften), wenn nach links geshiftet wird. Wird nach rechts geshiftet, kann es sein, daß eine binäre 1 nachgezogen wird, wenn die zugrundeliegende Zahl vorzeichenbehaftet und negativ ist (höchstwertiges Bit ist gesetzt). Dies muß aber nicht so sein! In allen anderen Fällen werden binäre Nullen nachgezogen.

  Die Anzahl der zu verschiebenden Stellen darf nicht größer sein als die binäre Stellenzahl des jeweiligen Operanden.

  Beide Operanden müssen ganzzahlig sein; der rechte muß positiv sein!

- `<`, `>`, `<=` und `>=` vergleichen numerische Werte (ganzzahlig oder gebrochen oder Zeiger) auf *kleiner*, *größer*, *kleiner gleich* und *größer gleich*.

  Ein solcher Vergleich liefert einen Wert vom Typ `int`, und zwar `1`, wenn die zugrundeliegende Bedingung erfüllt ist; sonst `0`.

- `==` und `!=` vergleichen zwei Operanden auf Gleichheit beziehungsweise Ungleichheit. Das Ergebnis des Vergleichs hat den Wert 1 (wenn erfüllt) oder 0 (nicht erfüllt).

  Achtung! Der Vergleich auf Gleichheit darf nicht mit dem Operator `=`, sondern nur mit `==` durchgeführt werden! Einer der häufigsten und manchmal gemeinsten Fehler in *C* ist das einfache Gleichheitszeichen in diesem Fall. Die Gemeinheit kommt daher, daß der Compiler in den meisten Fällen[23] stillschweigend die Zuweisung akzeptiert und keinen Fehler erkennt. Die Zuweisung selbst liefert wieder einen Wert, der dann anstelle des logischen Ausdrucks auf ungleich `0` verglichen wird.

  Bei Gleitkommazahlen ist der Vergleich auf Gleichheit oder Ungleichheit gefährlich, wenn die Werte vorher berechnet wurden. Durch unbeabsichtigte Rundungsfehler wird das exakte Ergebnis nur in den seltensten Fällen getroffen und ein anschliessender Vergleich ist deshalb mit Vorsicht zu genießen. Anstatt

  ```
  double a, b;
  a = ...;
  b = ...;
  if( a==b ) ...
  ```

  sollte man stattdessen lieber eine Konstruktion der Art

  ```
  #include <math.h>
  ...
  double a, b;
  a = ...;
  b = ...;
  if( fabs( a, b )<1.0E-10 ) ...
  ```

  verwenden.

- `&` liefert die binäre *UND*-Verknüpfung seiner beiden Operanden. Im Ergebnis ist ein Bit also genau dann gesetzt, wenn das entsprechende Bit in beiden Operanden gesetzt ist.

  Nur zulässig für ganzzahlige Operatoren.

- `^` liefert analog dazu die Verknüpfung mit dem *exklusiven Oder*: ein Bit im Ergebnis ist genau dann gesetzt, wenn es in genau einem der Operanden gesetzt ist, aber nicht in beiden.

  Nur zulässig für ganzzahlige Operatoren.

- `|` verknüpft zwei Operanden bitweise mit dem *Oder*. Ein Ergebnisbit ist genau dann gesetzt, wenn es zumindest in einem Operanden gesetzt ist.

  Nur zulässig für ganzzahlige Operatoren.

[23]nämlich dann, wenn auf der linken Seite ein Wert steht, dem etwas zugewiesen werden kann, ein sogenannter *lvalue*. Einige Compiler sind so gnädig und geben in einigen Fällen wenigstens eine Warnung aus, wie etwa „`possibly incorrect assignment`“.

- `&&` und `||` sind das *logische Und* beziehungsweise das *logische Oder*. Sie liefern dann den Wert 1, wenn zumindest einer (`||`) beziehungsweise beide (`&&`) Operanden ungleich `0` sind.

  C hat eine nette Eigenart: Vergleiche mit `&&` und `||` werden prinzipiell von links nach rechts bewertet, und zwar nur soweit, bis das Ergebnis des (Teil-)Ausdrucks feststeht (*short circuit evaluation*). Dies sorgt für eine schnellere Auswertung zur Laufzeit. Andererseits kann es vorkommen, daß dadurch Ausdrücke nicht bewertet werden. Insbesondere Funktionen mit Seiteneffekten fallen manchmal unangenehm auf, wenn man nicht damit rechnet, daß der zweite Teil eines Und- bzw. Oder-Vergleichs gar nicht ausgeführt wird.

- `?:` ist ein Auswahloperator. Er hat als einziger Operator drei Operanden! Der erste Operand steht vor dem `?` und wird als logischer Ausdruck bewertet (also auf *ungleich 0* getestet). Die beiden anderen Operanden stehen vor und nach dem Doppelpunkt `:`. Wenn die Bedingung (der erste Operand) als *logisch wahr* bewertet wurde, dann wird der Wert des zweiten Operanden berechnet und geliefert, ansonsten der Wert des dritten Operanden.

  Sowohl der Typ des zweiten als auch des dritten Operanden muß in der Umgebung des `?:`-Operators verwendet werden können.

  Man kann damit das Programmstück

  ```
  if( i<j )
      a = 123;
  else
      a = 456;
  ```

  ersetzen durch:

  ```
  a = i<j ? 123 : 456;
  ```

  Ein schönes Beispiel dafür sind die `MAX`- und `MIN`-Makros, die man sich so definieren kann:

  ```
  #define MIN(a,b)    ( (a)<(b) ? (a) : (b) )
  #define MAX(a,b)    ( (a)>(b) ? (a) : (b) )
  ```

  Siehe dazu auch **5 Der Praeprozessor** auf Seite 17.

- Die Zuweisung `=` ist ein Operator wie andere auch und liefert neben ihrer eigentlichen Tätigkeit, dem Zuweisen, auch noch ein Ergebnis, nämlich den zugewiesenen Wert. Dadurch kann man eine Zuweisung gleich in einem umgebenden Ausdruck weiterverwenden.

  Mit diesem zusätzlichen Ergebnis einer Zuweisung ist es ähnlich wie mit dem Rückgabewert einer Funktion: man kann es verwenden, man muß aber nicht.

  Der zugewiesene Wert muß natürlich entweder den gleichen Typ haben wie das Objekt, an das zugewiesen wird, oder zumindest entsprechend konvertiert werden können.

  Beispiel:

  ```
  i = ( j = 5 ) + 1;
  ```

  weist an `j` den Wert `5` zu; dann wird zu dem zugewiesenen Wert nochmals `1` addiert und die resultierende `6` in die Variable `i` geschrieben. Die Zuweisung an `i` liefert den Wert `6` als Ergebnis, der dann nicht weiter verwendet wird.

- Es gibt auch noch spezielle Zuweisungen (`+=`,`-=`, `*=`, `/=`, `%=`, `&=`, `^=`, `|=`, `<<=`, `>>=`), die ihren linken Operanden mit ihrem rechten verknüpfen. Die Art der Verknüpfung entspricht dem jeweiligen Operator ohne `=`, also `+`, `-`, `*`, `/`, `%`, `&`, `^`, `|`, `<<` und `>>`.

  Die Operatoren `%=`, `&=`, `^=`, `|=`, `<<=` und `>>=` sind wiederum nur für ganzzahlige Operanden zulässig.

  Beispielsweise kann man

  ```
  i = i + 5;
  ```

  ersetzen durch:

  ```
  i += 5;
  ```

  Besonders günstig ist dies, wenn der Wert auf der linken Seite ein komplizierter Ausdruck ist, der entsprechend Rechenzeit zur Auswertung benötigt.

  Beispielsweise soll in einer Quelldatei eine Variable `_fehler` als `static` vereinbart sein. Auf sie kann man von anderen Quelldateien aus nur mit Hilfe einer Funktion `fehlerp()` zugreifen, welche die Adresse der Variablen liefert:

  ```
  static int _fehler = 0;

  int *fehlerp( void )
  {
      return &_fehler;
  }
  ```

  Will man die Variable jetzt von einem anderen Quelltext aus um den Wert 10 erhöhen, dann könnte man das so tun:

  ```
  *fehlerp() = *fehlerp() + 10;
  ```

  Dadurch werden aber zwei Funktionsaufrufe ausgeführt!

  Schneller läuft mit Sicherheit diese Programmversion:

  ```
  *fehlerp() += 10;
  ```

- Der Kommaoperator `,` hat zwei Operanden: einen links vom Komma und einen rechts davon. Der erste Operand wird zuerst ausgeführt; das Ergebnis, welches er liefert, wird verworfen. Dann wird der zweite (rechte) Operand bewertet und dessen Ergebnis geliefert.

  Beispiel: Der Ausdruck

  ```
  ergebnis = sin(x) + sqrt( sin(x) );
  ```

  muß zweimal die Funktion `sin()` aufrufen, die erfahrungsgemäß viel Zeit vertrödelt. Mit dem Ausdruck

  ```
  tmp = sin(x), ergebnis = tmp + sqrt( tmp );
  ```

  erreicht man das Gleiche[24].

  Der Kommaoperator garantiert zwar, daß zuerst der linke Operand und dann der rechte ausgewertet wird. Dies hat jedoch nichts mit der Auswertungsreihenfolge der Parameter von Funktionen zu tun! Deren Berechnungsreihenfolge ist nicht definiert, auch wenn sie durch Kommata getrennt werden.

[24] Natürlich hätte man stattdessen auch zwei komplette Anweisungen nehmen können.

12 Steueranweisungen

Mit den Steueranweisungen einer Programmiersprache kann man den Programmlauf steuern, also den linearen Ablauf (erst die erste Anweisung ausführen, dann die zweite, und so fort) unterbrechen.

Die Kontrollstrukturen in *C* sind die Fallunterscheidungen

- if(...), else if(...), else
- switch(...)...

und die Schleifenkonstruktionen

- while(...) ...
- do ... while(...)
- for(...; ...; ...) ...

sowie die `goto`-Anweisung.

Alle Kontrollstrukturen (außer `switch` und `goto`) beziehen sich normalerweise entweder auf eine Anweisung, die dann mit einem Semikolon (`;`) beendet wird, oder auf einen Block, also eine Folge von beliebig vielen Anweisungen (notfalls auch keine!), die von geschweiften Klammern eingeschlossen sind. In letzterem Falle gehört weder vor noch hinter den Block ein Semikolon!

12.1 if, else if, else

Die `if`-Anweisung enthält in dem Klammerpaar einen logischen oder numerischen Ausdruck, der auf *ungleich 0* geprüft wird. Nach dem Klammerpaar folgt eine Anweisung oder ein Block, in dem mehrere Anweisungen stehen können. Folgt kein Block in geschweiften Klammern, dann gehört nur die nächste folgende Anweisung zum `if`.

Die Anweisung oder der Block werden nur ausgeführt, wenn die Bedingung *logisch wahr*, also ungleich `0`, ist.

`else if` ist analog zum `if`. Dahinter und auch hinter `else` folgt ebenfalls wieder eine Anweisung oder ein Block.

`else if` und `else` beziehen sich immer auf das letzte davorstehende, noch offene `if`.

Von allen zusammengehörenden `if` — `else if` — `else` - Teilen wird immer nur ein Anweisungsteil ausgeführt. Danach wird mit der ersten folgenden Anweisung das Programm fortgesetzt.

Beispiel:

```
/* Dieses puts() wird nur ausgefuehrt, wenn i<j ist:    */
if( i<j ) puts( "i ist kleiner als j" );

/* Eines der naechsten drei puts() wird ausgefuehrt:    */
if( a>b ) puts( "a ist groesser als b" );
else if( a==b ) puts( "a ist gleich b" );
else puts( "a ist kleiner als b" );

/* Diese Anweisung wird in jedem Fall ausgefuehrt:      */
a = b;
```

Sowohl der `else if`- als auch der `else`-Teil dürfen fehlen. Dagegen dürfen mehrere `else if`-Teile vorkommen, aber nur ein `else`.

12.2 switch

Mit der `switch`-Anweisung kann man in Abhängigkeit von einem numerischen Ausdruck zu einer bestimmten Anweisung springen.

Direkt nach dem Schlüsselwort `switch` folgt in runden Klammern ein numerischer Ausdruck, der ausgewertet wird. Danach folgt ein Block von Anweisungen. Vor jeder Anweisung in diesem Block dürfen eine oder mehrere `case`-Marken stehen. Eine solche Marke besteht aus dem Schlüsselwort `case`, einer ganzzahligen Konstanten und einem Doppelpunkt. Die `default`-Marke steht für alle Werte, zu denen kein `case` angegeben ist und kann auch weggelassen werden.

Wird eine Marke angesprungen, dann hindern folgende Marken nicht den Programmfluß. Dies kann mit einer `break`-Anweisung verhindert werden.

Beispiel:

```
/* koswitch.c 30. 7.95 kw
 */

int main()
{
  int i;

  /* Einlesen einer ganzen Zahl von der Tastatur:
   */
  scanf( "%d", &i );

  /* Je nach dem, welchen Wert i hat, zu einer der
   * case-Marken springen:
   */
  switch( i )
  {
    case 1:
      puts( "i ist 1" );
    case 2:
    case 3:
      puts( "i ist 1, 2 oder 3" );
      break;
```

```
    case 9:
      puts( "i ist 9" );
    default:
      puts( "was weiss denn ich" );
  }

  puts( "fertig." );  /* Schlusstext.
                       */
  return 0;
}
```

Dieses Programm liest eine Zahl (mit **sscanf()** (Seite 224)) ein und verzweigt je nach dem Wert zu einer Anweisung in dem `switch`-Block.

Wurde der Wert `1` gelesen, dann springt das Programm zu der ersten `puts( ¨i ist 1¨ )`-Anweisung. Die nach `case 3:` folgende Anweisung wird aber auch noch ausgeführt; erst dann wird mit `break` die `switch`-Konstruktion verlassen und mit `puts( ¨fertig.¨ );` fortgefahren.

Ebenso wird im Falle von `i==9` auch der `default`-Teil mit ausgeführt.

Achtung! Wie in jedem Block dürfen auch in dem Anweisungsblock einer `switch`-Anweisung Variablen vereinbart werden. Allerdings dürfen diese Variablen bei ihrer Vereinbarung **NICHT** initialisiert werden! Ob ein Compiler dies allerdings als Fehler erkennt, ist fraglich. In jedem Fall wird eine solche Initialisierung nie ausgeführt!

12.3 while

Die `while`-Anweisung wiederholt einen Anweisungsteil (eine Anweisung oder einen Block) eventuell mehrmals. Vor jedem erneuten Durchlauf wird eine Laufbedingung überprüft. Ist diese nicht erfüllt, dann wird mit der Ausführung der Schleife abgebrochen. Diese Bedingung wird immer nur am Schleifenanfang getestet.

Die `while`-Schleife ist also eine abweisende Schleife: Wenn die Laufbedingung schon beim ersten Auswerten nicht erfüllt ist, dann wird der Anweisungsteil gar nicht ausgeführt.

Innerhalb der Schleife kann man aber auch gezielt mit einer `break`-Anweisung die Schleife ganz abbrechen, oder mit `continue` den Rest des Anweisungsteils überspringen und gleich zum Prüfen der Bedingung zurückkehren.

Beispiele:

```
while( i ) --i;
```

zählt die Variable `i` so lange herunter, bis sie `0` ist.

```
while( --i );
```

macht fast das gleiche; die Anweisung hinter `while()` ist die leere Anweisung[25].

[25] Der Unterschied liegt nur darin, daß in der letzteren Version für den Fall, daß die Variable `i` vor Eintritt in die Schleife schon den Wert `0` hat, `i` erstmal auf `-1` gesetzt und dann solange heruntergezählt wird, bis sie wieder `0` ist. Dies ist mathematisch natürlich nicht möglich, aber durch einen Überlauf in der Arithmetik meistens doch der Fall.

```
while( (c=fgetc(f))!=EOF )
{
    printf( "c ist %d\n", c );
    if( c==26 ) break;
    if( isprint(c) )
        {
            printf( "%c\n", c );
            continue;
        }
    printf( "c ist nicht druckbar!\n" );
}
```

Hier wird die Schleife vorzeitig mit `break` verlassen, wenn der Wert `26` gelesen wurde.

Mit der `continue`-Anweisung wird zum Prüfungsteil der Schleife gesprungen, ohne den Rest des Schleifenkörpers zu beachten.

12.4 do while

Die `do`...`while`-Schleife ist im Gegensatz zu `while` die nicht abweisende Schleife. Hier wird der Anweisungsteil zuerst ausgeführt; dann wird die Bedingung überprüft.

Beispiel:

```
do
    {
        printf( "Bitte einen positiven Wert eingeben:" );
        scanf( "%d", &wert );
    }
while( wert<0 );
```

`break` und `continue` sind identisch zu verwenden wie in der `while`-Schleife.

12.5 for

Die `for`-Schleife hat die Form

`for(` *initialisierung*`;` *laufbedingung*`;`*reinitialisierung* `)`*anweisung*

wobei *anweisung* eine einzelne Anweisung (mit Semikolon (`;`) beendet!) sein kann oder ein ganzer Block, der mit {} geklammert wird und aus beliebig vielen Anweisungen bestehen kann (hinter dem Block ist dann kein Semikolon (`;`) nötig).

Alles, was im Teil *initialisierung* steht, wird vor Beginn der Schleife ausgeführt. Man kann in *C* im Gegensatz zu anderen Sprachen auch mehrere Anweisungen angeben, die vor der Schleife ausgeführt werden sollen; die einzelnen Anweisungen sind dann mit Kommata (`,`) getrennt. Der Teil *laufbedingung* wird vor jedem Schleifendurchlauf ausgeführt und geprüft, ob er den Wert *wahr* oder *falsch* liefert. Wenn er *wahr* liefert, dann wird die Schleife mindestens noch einmal durchlaufen, ansonsten nicht mehr; dann wird mit der ersten Anweisung nach der Schleife weitergemacht.

Nach jedem Schleifendurchlauf, aber vor der nächsten Prüfung von *laufbedingung*, wird der Teil *reinitialisierung* durchgeführt. Dies kann man wie hier zum Erhöhen der Schleifenzählers nutzen. Es können auch mehrere Anweisungen stehen, die an dieser Stelle ausgeführt werden sollen; diese sind dann wieder mit Kommata zu trennen.

Jeder Teil in der for-Schleife kann auch fehlen (außer for() natürlich).

break und continue sind identisch zu verwenden wie in der while-Schleife. Der einzige Unterschied ist der, daß bei der continue-Anweisung zuerst die Reinitialisierung ausgeführt wird.

Beispiel:

```
int f[10];
int i;
for( i=0; i<10; i++ ) f[i] = 0;
```

ist identisch mit:

```
int f[10];
int i;
for( i=0; i<10; ) f[i++] = 0;
```

Endlosschleifen kann man wahlweise mit while, do ... while oder for erzeugen. Üblich sind die beiden Konstruktionen while(1) ... und for(;;) .... Der Quelltext wird schön lesbar, wenn man einmalig (ggf. in einer Headerdatei) sich eine Endlosschleife definiert:

```
#define     forever for(;;)
```

und dann die Endlosschleife mit forever ... schreibt.

12.6 goto

Die goto-Anweisung kann man verwenden, um direkt an eine bestimmte Stelle innerhalb der aktuellen Funktion zu springen.

Dazu muß man die betreffende Stelle mit einer Marke (*label*) versehen. Das ist ein Name, gefolgt von einem Doppelpunkt:

```
oben:
    scanf( "%d", &i );
    if( i<10 ) goto oben;
```

(Eine Marke muß vor ihrer Verwendung nicht deklariert werden; das goto darf trotzdem vor der Marke stehen.)

Diese Anweisung ist nur selten sinnvoll, da sie das Programm unübersichtlich und fehlerträchtig macht. Sie hat dann ihre Berechtigung, wenn in tief geschachtelten Blöcken innen irgendwo ein Fehler festgestellt wird und die Funktion aber nicht gleich mit return verlassen werden kann, weil beispielsweise noch irgendetwas aufzuräumen ist. Dann kann man zu einer Anweisung innerhalb der Funktion springen, ab der die nötigen Arbeiten durchgeführt werden. Mit break dagegen könnte man immer nur die innerste Schleife verlassen.

Allgemein anerkannte, dringende Empfehlung: goto sparsam verwenden!

Achtung! Vor Sprungmarken wird oft und gerne eine return-Anweisung vergessen. Dann geht der normale Programmlauf in den Funktionsteil hinein, der beispielsweise zur Fehlerbehandlung gedacht ist.

Eine weitere Fehlerquelle sind Sprünge in einen Block mit lokalen Variablen, da in diesem Fall die Initialisierung der Variablen unterbleibt.

Über die Grenzen von Funktionen hinweg kann man damit nicht springen. Dies ist mit der Sprachdefinition von *C* nicht direkt möglich; es gibt aber genormte Funktionen dafür in der Standardbibliothek. Siehe dazu `setjmp()` (Seite 221) und `longjmp()` (Seite 205).

13 Rekursion

Mit Rekursion verhält es sich ähnlich wie mit Zeigern: Eigentlich nicht schwer zu verstehen, aber in der Praxis doch immer wieder undurchsichtig und fehleranfällig[26].

Rekursion heißt nur, daß sich eine Funktion selbst wieder aufruft.

Meist geschieht dies direkt, indem in einer Funktion ein Aufruf von sich selbst steht. Es kann aber auch so geschehen, daß eine Funktion `a()` eine weitere Funktion `b()` aufruft, und die wiederum ruft `a()` auf.

In *FORTRAN* ist Rekursion ausdrücklich verboten, auch wenn es manchmal (mit Einschränkungen) geht. In *Pascal* ist Rekursion ebenso wie in *C* zulässig und durchaus sinnvoll einzusetzen.

Ein beliebtes Beispiel für Rekursion ist die Berechnung der Fakultät einer gegebenen Zahl. Die Fakultät $l!$ einer positiven Zahl l kann man (nicht rekursiv, also iterativ) so definieren:

$$l! = 1 \times 2 \times 3 \times \cdots \times l = \prod_{i=1}^{l} i \tag{1}$$

Die Fakultät von l ist also das Produkt aller Zahlen von 1 bis l. Die Fakultät von 0 ist vereinbarungsgemäß 1.

Die Umsetzung in eine *C*-Funktion und ein kleines Testprogramm ist einfach:

```
/* refaki.c 30. 7.95 kw
 */

#include <stdio.h>

/* berechnet die Fakultaet einer Zahl l:
 */
unsigned long int fak( unsigned int l )
{
  int                    i;
  unsigned long int      ergebnis;

  ergebnis = 1;
  for( i=2; i<=l; i++ )
  {   ergebnis *= i;
  }
  return ergebnis;
}

/* Testprogramm fuer fak():
                         */
int main()
{
  unsigned int    l;

  for( l=0; l<10; l++ )
```

[26] Ähnlich wie `goto` ist das Thema Rekursion gut geeignet, alte Feindschaften zu pflegen und neue aufbrechen zu lassen. Mehr dazu in [c't Rek.].

```
    printf( "fak(%u) ist %lu.\n", l, fak(l) );
}
```

Für die Definition der Fakultät wird aber statt Gleichung (1) oft eine rekursive Vorschrift angegeben:

$$l! = \begin{cases} 1 & \text{für } l = 0 \\ (l-1)! \times l & \text{sonst} \end{cases} \tag{2}$$

Diese Definition verwendet selbst den Term, den sie definieren soll. Da aber auch eine rekursive Definition sich nicht selbst an den Haaren aus dem Sumpf ziehen kann, ist eine Sonderbehandlung für einen Startwert nötig (hier für den Wert $l = 0$).

Rekursive Definitionen lassen sich nun in *C* ebenso wie in *Pascal* direkt in Funktionen umsetzen:

```
/* refakr.c 30. 7.95 kw
 */

#include <stdio.h>

/* berechnet die Fakultaet einer Zahl l (rekursiv):
 */
unsigned long int fak( unsigned int l )
{
  if( 0==l )        /* Sonderbehandlung        */
    return 1;
  else              /* Rekursion               */
    return fak( l-1 )*l;
}

/* Testprogramm fuer fak():
 */
int main()
{
  unsigned int    l;

  for( l=0; l<10; l++ )
    printf( "fak(%u) ist %lu.\n", l, fak(l) );
}
```

Diese Implementation von `fak()` kommt ohne Schleifen aus. Bei jedem Aufruf mit einem Argument ungleich `0` ruft sich `fak()` selbst auf mit einem um `1` verkleinerten Argument und multipliziert das Ergebnis des Funktionsaufrufs mit seinem eigenen Argument `l`. Die mit dem um `1` erniedrigten Argument aufgerufene Funktion `fak()` vergleicht ihr eigenes Argument wieder mit `0`, ruft sich gebenenfalls nochmal selbst auf, und so weiter.

Beim Schreiben von rekursiven Algorithmen muß man gut darauf achten, daß die Aufruffolge beizeiten abbricht (so wie im obigen `fak()` beim Erreichen der `0`). Ansonsten geht das Programm formal in eine Endlosschleife über, da jeder Funktionsaufruf wieder einen weiteren nach sich zieht. In der Praxis kann ein solches Programm aber nicht endlos laufen, da bei jedem Funktionsaufruf irgendwo die Programmadresse hinterlegt wird, an der das Programm nach dem Abarbeiten der Funktion fortgesetzt werden soll (Rücksprungadresse). Dieser Speicherbereich (meist Stack genannt) wird natürlich irgendwann voll, wenn eine endlose Rekursion eintritt, und das Programm oder gar der Rechner verabschieden sich früher oder später[27].

Wenn eine rekursiv aufgerufene Funktion lokale automatische Variablen hat, dann hat jede Aufrufebene einen eigenen Satz dieser Variablen. Den Gegensatz dazu bilden Variablen, die mit dem Schlüsselwort `static` vereinbart sind. Diese `static`-Variablen existieren auch bei rekursiven Funktionen nur einmal (**8.6 Attributangaben** auf Seite 50). Dies liegt daran, daß mit jedem Funktionsaufruf ein eigener *stack frame* angelegt wird, in dem die automatischen Variablen liegen; jede Rekursionsebene erhält ihren eigenen stack frame (siehe dazu auch **9 Speichermodell eines Prozesses** auf Seite 73).

Beispiel:

```
/* rewieof1.c 30. 7.95 kw
 */

void rekursiv( int wieoft )
{
  int     i;

  if( wieoft>0 )
  {
    i = wieoft;
    rekursiv( wieoft-1 );
    printf( "i ist %d, wieoft ist %d\n", i, wieoft );
  }
}

int main()
{
  rekursiv( 10 );
}
```

Die Funktion `rekursiv()` wird von `main()` mit dem Parameter `10` aufgerufen. `rekursiv()` ruft sich dann, wenn ihr Parameter größer ist als `0`, mit einem um `1` kleineren Argument wieder selbst auf. In der lokalen Variablen `i` wird aber vor dem Aufruf der aktuelle Wert des Parameters `wieoft` festgehalten. Nach dem Selbstaufruf werden der in `i` gespeicherte Wert und der Parameter `wieoft` ausgegeben. Die Programmausgabe sieht so aus:

```
i ist 1, wieoft ist 1
i ist 2, wieoft ist 2
i ist 3, wieoft ist 3
i ist 4, wieoft ist 4
i ist 5, wieoft ist 5
i ist 6, wieoft ist 6
i ist 7, wieoft ist 7
i ist 8, wieoft ist 8
i ist 9, wieoft ist 9
i ist 10, wieoft ist 10
```

[27] Je nach System und Compiler bricht das Programm in einem solchen Fall kontrolliert mit einer Fehlermeldung ab, oder es stürzt unkontrolliert ab. Auf vernünftigen Systemen kann man zumindest durch Compileroptionen zwischen diesen beiden Möglichkeiten wählen (was dann bei einer Überprüfung etwas Laufzeiteffizienz kostet), oder (noch besser) das Betriebssystem kümmert sich automatisch darum.

Jeder Aufruf der Funktion `rekursiv()` erzeugt also eine neue Variable `i`, da Aufrufe von `rekursiv()` nach der Zuweisung an `i` die Variable nicht mehr verändern.

Man kann aber `i` stattdessen auch als `static` vereinbaren. Dann verfügen alle Aufrufebenen von `rekursiv()` nur über eine gemeinsame Variable, und jede Zuweisung an `i` verändert diese Variable auch für die anderen Aufrufebenen sichtbar:

```
/* rewieof2.c 30. 7.95 kw
 */

void rekursiv( int wieoft )
{
  static int      i;

  if( wieoft>0 )
  {
    i = wieoft;
    rekursiv( wieoft-1 );
    printf( "i ist %d, wieoft ist %d\n", i, wieoft );
  }
}

int main()
{
  rekursiv( 10 );
}
```

Der einzige Unterschied zum vorhergehenden Beispiel ist das Schlüsselwort `static` bei der Definition von `i`.

Durch den gemeinsamen Zugriff auf eine einzige Variable `i` verändert sich bei einer Zuweisung an sie der Wert „aller“ Variablen `i`; die Ausgabe sieht dann so aus:

```
i ist 1, wieoft ist 1
i ist 1, wieoft ist 2
i ist 1, wieoft ist 3
i ist 1, wieoft ist 4
i ist 1, wieoft ist 5
i ist 1, wieoft ist 6
i ist 1, wieoft ist 7
i ist 1, wieoft ist 8
i ist 1, wieoft ist 9
i ist 1, wieoft ist 10
```

Bei vielen Problemstellungen hat man die freie Auswahl zwischen rekursiven und iterativen Algorithmen. Dann lassen sich nur wenige Programmierer freiwillig auf die Rekursion ein. Oft ist aber eine rekursive Datenstruktur vorgegeben, wie etwa verkettete Listen oder Binärbäume. Dann ist die rekursive Denkweise in aller Regel weniger aufwendig als die iterative. Wenn die Daten keine Verzweigungen aufweisen (lineare Listen), dann kommt man oft noch leicht ohne Rekursion aus. Hat man beispielsweise eine einfach verkettete Liste mit `double`-Werten definiert, dann kann man die Werte in der Liste mit einer iterativen Funktion so ausgeben:

```
/* relist.c 30. 7.95 kw
 */

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

/* Struktur eines Listenelements:
 */
struct liste
{
  double          wert;
  struct liste   *next;
};

/* iterative Ausgabe einer ganzen Liste:
 */
void zeige( struct liste *l )
{
  while( l!=NULL )
  {
    printf( "%f\n", l->wert );
    l = l->next;
  }
}

/* Das Hauptprogramm erzeugt eine Liste mit 3 Elementen und
 * laesst sie anzeigen:
 */
main()
{
  struct liste
    *l,               /* Anfang der Liste          */
    *lp;              /* wandert die Liste entlang */

  /* Speicher fuer erstes Element allokieren:         */
  lp = l = malloc( sizeof(struct liste) );

  /* Wert und Nachfolger eintragen:                   */
  lp->wert = 3.1415;
  lp->next = malloc( sizeof(struct liste) );
  lp = lp->next;

  /* Wert und Nachfolger eintragen:                   */
  lp->wert = 4.12;
  lp->next = malloc( sizeof(struct liste) );
  lp = lp->next;

  /* Wert und Nachfolger eintragen:                   */
  lp->wert = 7.77;
  lp->next = NULL;    /* Kein Nachfolger, NULL ist
                       * Markierung fuer das
                       * Listenende.
                       */
  zeige( l );         /* Anzeigen lassen
```

```
                        */
}
```

Die Funktion `zeige()` gibt in dem obigen Beispielprogramm eine einfach verkettete Liste mit Hilfe einer `while`-Schleife aus. Diese Funktion ließe sich auch durch eine rekursive Version ersetzen:

1) das erste Listenelement ausgeben,
2) sich selbst aufrufen, um den Rest der Liste auszugeben.

Dies sieht in *C* dann so aus:

```
/* relistr.c 30. 7.95 kw
 */

/* rekursive Ausgabe einer ganzen Liste:
 */
void zeige( struct liste *l )
{
  if( l!=NULL )
  {
    printf( "%f\n", l->wert );
    zeige( l->next );
  }
}
```

Iterative und rekursive Version sind im wesentlichen gleichwertig. Rekursive Algorithmen brauchen aber bei tiefen Verschachtelungen ausreichend Stack (für Aufrufparameter, eventuelle lokale Variablen und die Rücksprungadresse, siehe **9 Speichermodell eines Prozesses** auf Seite 73), der für die iterative Lösung hier nicht nötig ist. Die Größe des zur Verfügung stehenden Stacks läßt sich bei vielen System beim Übersetzen, Binden oder Laden des Programms festlegen.

Bei rekursiven Algorithmen kann man oft verblüffende Effekte sehr einfach erreichen. Will man beispielsweise in der iterativen Version von `zeige()` die umgekehrte Ausgabereihenfolge erzwingen, dann muß man einen ziemlichen Aufwand treiben. Die rekursive Version dagegen kann man leicht dazu überreden: Die Funktion muß nur

1) erst sich selbst aufrufen, um den Rest der Liste auszugeben,
2) dann das erste Listenelement ausgeben.

Die Änderung am Quelltext ist minimal:

```
/* relistrr.c 30. 7.95 kw
 */

/* umgedrehte rekursive Ausgabe einer ganzen Liste:         */
void zeige( struct liste *l )
{
  if( l!=NULL )
  {
```

```
    zeige( l->next );
    printf( "%f\n", l->wert );
  }
}
```

Soweit zu einfach verketteten Datenstrukturen, die sich auch iterativ relativ bequem behandeln lassen. Besitzt eine Datenstruktur Verzweigungen (Bäume), dann kann man die meisten Operationen rekursiv wesentlich einfacher durchführen.

Als Beispiel für Bäume dient eine einfache Verwaltung von eingelesenen Gleitkommazahlen. Im Hauptprogramm hat man ein kleines Menü, mit dessen Hilfe man neue Werte eingeben oder alle gespeicherten Werte sich anzeigen lassen kann.

Ein Binärbaum besteht aus einem Zeiger auf den Wurzelknoten (oder `NULL`, wenn der Baum anfangs noch leer ist). Der Wurzelknoten kann einen Wert speichern (hier die Gleitkommazahl) und auf einen linken sowie auf einen rechten Nachfolger verweisen, ebenso jeder weitere Knoten. Ein Knoten kann als linken und/oder rechten Nachfolger eine `NULL` eingetragen haben; dann existiert eben derjenige Nachfolger nicht. Ein Knoten, der weder einen linken noch einen rechten Nachfolger hat, heißt *Blatt*.

Jeder Knoten kann wieder als Wurzel eines kleineren Teilbaums betrachtet werden.

Die Daten in einem solchen Baum werden nun so eingetragen, daß der Wert eines jeden Knotens größer ist als der eines eventuellen linken Nachfolgers, aber kleiner als der Wert eines eventuellen rechten Nachfolgers.

Beim Einfügen in den Baum muß man den Wert der Wurzel betrachten und entsprechend links oder rechts davon eintragen. Angenommen, man muß links eintragen, weil der neue Wert kleiner ist als der Wert der Wurzel, dann muß man den linken Nachfolger der Wurzel betrachten und links oder rechts davon eintragen, und so weiter. Hat ein Knoten auf der Seite, auf der man eintragen muß, keinen Nachfolger, dann hängt man dort als neuen Nachfolger das einzutragende Element ein.

Um einen Wert im Baum zu suchen, kann man ebenfalls bei der Wurzel beginnen, und mit links/rechts-Entscheidungen bis zur gesuchten Stelle vordringen. Trifft man unterwegs auf `NULL`, dann existiert das gesuchte Element nicht.

Ein Binärbaum läßt sich leicht sortiert ausgeben, wenn die Funktion zur Ausgabe des Baums einfach sich selbst mit dem linken Teilbaum aufruft (der nur kleinere Werte als die Wurzel enthält), dann den Wert der Wurzel ausgibt, und sich dann mit dem rechten Teilbaum selbst aufruft. Dieses Vorgehen heißt *in order*-Durchlauf.

Sowohl die Funktion `einfuege_baum()` zum Einfügen eines neuen Elements in den Baum als auch `zeige_baum()` zur sortierten Anzeige des Baums sind rekursiv gelöst:

```
/* rebaum.c 30. 7.95 kw
 *
 * Testprogramm fuer einfache Binaerbaeume.
 *
 * Das Programm bietet ein kleines Menue an, mit dessen
 * Hilfe man Zahlen eingeben kann, alle eingegebenen Zahlen
 * sortiert wieder ausgeben, oder das Programm beenden kann.
 *
 * Ein Knoten in einem Baum besteht aus den zu speichernden
 * Daten (hier als Beispiel eine Gleitkommazahl), sowie
 * zwei Zeigern auf weitere Knoten (links, rechts
 * Der Baum wird nun so sortiert aufgebaut, dass alle Werte
```

```
 * in einem linken Nachfolger und dessen Nachfolgern klei
 * ner sind als der Wert jeden Knotens und alle Nachfolger
 * auf der rechten Seite groesser
 * Hat ein Knoten keinen linken oder rechten Nachfolger
 * dann hat (links) bzw. (rechts) den Wert NULL
 */

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

/* Datenstruktur fuer einen Knoten:
 */
struct baum
{
  double      wert;
  struct baum *links, *rechts;
};

/* Der eigentliche Baum besteht aus einem Zeiger auf das
 * erste Element (die "Wurzel") und den Nachfolgern.
 * Solange kein Element gespeichert ist, hat die Wurzel den
 * Inhalt NULL.
 */

struct baum    *wurzel = NULL;

void einfuege_baum( double zahl, struct baum **baum_p )
{
  /* an der aktuellen Stelle einfuegen, wenn dort
   * noch nichts steht:
     */
  if( NULL==(*baum_p)  )
  {
    /* Platz fuer einen Knoten holen ...
     */
    *baum_p = malloc( sizeof(struct baum) );
    if( NULL==(*baum_p) )
    {
      fprintf( stderr,
               "nicht genug Speicher!\n"
               );
      exit(3);
    }

    /* ... den Wert eintragen und die Nachfolger
     * zu NULL setzen:
     */
    (*baum_p)->wert   = zahl;
    (*baum_p)->links  = NULL;
    (*baum_p)->rechts = NULL;
  }
  else
  {
```

```
    /* an der aktuellen Stelle steht schon etwas!
     * Also links oder rechts im Baum eintragen, je nach
     * Groesse der einzutragenden Zahl:
     */
    if( zahl < (*baum_p)->wert )
      einfuege_baum( zahl, &((*baum_p)->links) );
    if( zahl > (*baum_p)->wert )
      einfuege_baum( zahl, &((*baum_p)->rechts) );
    else
      ; /* gleiche Zahl nicht nochmal eintragen!  */
  }
}

/* gibt einen Baum mit einem inorder-Durchlauf aus, also
 * sortiert:
 */
void zeige_baum( struct baum *baum )
{
  /* zeigt baum ueberhaupt auf ein Element?
   */
  if( baum!=NULL )
  {
    /* erst den linken Teilbaum ausgeben, ...       */
    zeige_baum( baum->links );

    /* ... dann den aktuellen Wert, ...             */
    printf( "Knoten hat den Wert %f\n", baum->wert );

    /* ... zuletzt den rechten Teilbaum:            */
    zeige_baum( baum->rechts );
  }
}

/* Bietet ein einfaches Menue an und liefert das
* entsprechende Zeichen zurueck:
                           */
int get_menue( void )
{
  char c;

  printf( " Bitte Auswahl eingeben:\n" );
  printf( " (E)infuegen, (Z)eigen, (A)bbruch\n" );
  do
  {
    c = toupper( getchar() );
  }
  while( c!='E' && c!='Z' && c!='A' );

  return c;
}

/* Kleines Testprogramm.
 */
int main()
```

```
{
  double      wert;
  int         auswahl;

  /* Scheife, bis 'A' eingegeben wird:
   */
  while( ( auswahl=get_menue() )!='A' )
  {
    switch( auswahl )
    {
      case 'E' :
        /* Einfuegen gewuenscht.
         * Wert von Tastatur lesen:
         */
        printf( "Bitte Wert eingeben: " );
        scanf( "%lf", &wert );

        /* und einfuegen:
         */
        einfuege_baum( wert, &wurzel );

        break;

      case 'Z' :
        /* Zeigen gewuenscht
         */
        zeige_baum( wurzel );

        break;

      default  :
        break;
    }
  }
}
```

Theoretisch läßt sich jede Rekursion durch eine mehr oder wenige aufwendige Iteration ersetzen. Ob dies allerdings sinnvoll ist, läßt sich pauschal nicht sagen. Die Umwandlung einer rekursiven Lösung in eine iterative kann sehr einfach sein, wenn nur wenige Daten beteiligt sind und die Daten insbesondere keine Verzweigungen aufweisen (wie beispielsweise lineare Listen, siehe das Beispiel oben). Hat die zugrundeliegende Datenstruktur dagegen Verzweigungen (Bäume), dann kann die Umwandlung eines rekursiven Algorithmus in einen iterativen schnell sehr aufwendig und unsinnig werden.

Viele rekursive Algorithmen sind bestechend einfach und effektiv. Die iterative Version eines Algorithmus ist in der Regel etwas schneller, da bei Rekursion für jede Aufrufebene Parameter und Rücksprungadresse gespeichert werden müssen und jeder Rücksprung ebenfalls etwas Zeit kostet. Insbesondere sehr kurze Funktionen werden dadurch relativ stark gebremst; bei länger rechnenden Funktionen ist der Anteil des Aufrufs an der Rechenzeit geringer. In vielen Fälle ist die rekursive Lösung leichter verständlich und leichter zu programmieren; dann sollte sie man auch nicht mutwillig durch eine iterative ersetzen.

Wenn man rekursive Datenstrukturen (Bäume, Listen) verwendet, dann sind entsprechend gebaute rekursive Funktion die natürliche Lösung. Viele klassische Algorithmen (*quick sort*) sind ebenfalls in der rekursiven Form einfacher und direkter.

Wem die Rekursion dagegen einfach suspekt und unsympathisch ist, den wird man ohnehin nie dazu überreden können[28].

[28] Wahlspruch einiger gezwungener *Pascal*- und *C*-Programmierer ist:
„Rekursief geht's meistens schief!“

14 Die Parameter von main()

14.1 Programmargumente

Die Funktion `main()`, die in *C* die Rolle des Hauptprogramms in anderen Sprachen spielt, hat zwei Parameter und gibt einen Wert vom Typ `int` zurück.

Der Rückgabewert signalisiert der aufrufenden Umgebung, ob der Programmlauf erfolgreich war. Ein Wert von `0` wird in aller Regel zurückgegeben, wenn alles halbwegs gut ging; eine `1` signalisiert: „Na ja, gerade eben noch" und Werte ab `2` bedeuten, daß das eigentliche Ziel des Programms nicht erreicht werden konnte. Was mit dem Rückgabewert genau passiert, legt der Standard nicht fest.

Die Parameter, mit denen `main()` aufgerufen wird, sind aus der Kommandozeile aufbereitet, mit der das Programm aufgerufen wird. Wenn die Kommandozeile nicht benötigt wird, dann kann man die beiden Parameter getrost weglassen.

Der erste der beiden Parameter ist vom Typ `int` und gibt die Anzahl der Strings an, auf die der zweite Parameter zeigt. Der zweite Parameter ist vom Typ `char **` beziehungsweise `char *[]`. Er zeigt auf ein Feld, in dem wiederum Zeiger auf Strings hinterlegt sind. Die Anzahl der hinterlegten Strings und damit die Anzahl der Zeiger in dem Feld sind im ersten Parameter enthalten. Der erste Parameter ist immer mindestens `1`, da der erste Zeiger im Feld auf den Namen des aufgerufenen Programms zeigen soll. Dies ist aber oft nur unter *Unix* der Fall. Bei den meisten anderen Systemen ist der erste String leer und enthält nur die abschließende Null.

Ab dem zweiten String stehen dann die Worte aus der Kommandozeile, mit der das Programm aufgerufen wurde.

Folgendes Programm pamain.c gibt seine Parameter auf dem Bildschirm aus:

```
/* Time-stamp: "03.10.02 08:14 pamain.c klaus@wachtler.de"
 *
 * Ausgabe aller Programmargumente
 */

main( int n, char **s )
{
  /* Schleifenzaehler vereinbaren:
   */
  int     i;

  /* Schleife von i=0 bis (n-1) in Einerschritten:
   */
  for( i=0; i<n; i++ )
  {
    /* Ausgeben von i und dem i-ten String der
     * Kommandozeile:
     */
    printf( "Der %d. Parameter ist [%s]\n", i, s[i] );
  }
}
```

Wird es mit der Kommandozeile

```
abc defg
```

aufgerufen, dann gibt es folgendes aus:

```
Der 0. Parameter ist [pamain]
Der 1. Parameter ist [abc]
Der 2. Parameter ist [defg]
```

Die erste Zeile kann je nach dem verwendeten System auch anders aussehen.

14.2 Environmentvariablen

Unter allen gängigen Betriebssystemen gehört zu einem laufenden Prozeß ein sogenanntes *environment*; das ist nichts weiter als eine Sammlung von Variablen (Umgebungsvariablen, Environmentvariablen).

Jede dieser Variablen besteht aus einem Namen und einem Wert, wobei es keine verschiedenen Datentypen gibt. Vielmehr ist der Wert jeder vorhandenen Variable nur eine Zeichenkette.

Wenn auf einem konkreten System Environmentvariablen existieren, dann hat man damit neben den **Programmargumenten** (Seite 106) eine weitere Möglichkeit, Informationen an ein Programm zu übergeben.

Das ist sinnvoll für Informationen, die sich nicht ständig ändern und hautpsächlich für mehrere Prozesse gelten, wie beispielsweise der Name des aktuellen Benutzers (`USER`), Suchpfade für zu startende Programme (`PATH`), Homeverzeichnis des Benutzers (`HOME`), bevorzugter Drucker (`PRINTER`), und ähnliches. Anstatt solche Informationen bei jedem Programmstart angeben zu müssen, definiert man entsprechende Environmentvariablen. Diese bleiben während der gesamten Arbeitssitzung erhalten (sofern sie nicht explizit geändert werden) und stehen damit allen Programmen gleichermaßen zur Verfügung.

Jedes Programm kann die Variablen auswerten, oder sie ignorieren. In der Regel erben neu gestartete Programme alle Environmentvariablen ihres Aufrufers, ohne daß dazu besondere Vorkehrungen nötig wären.

Um in einem laufenden Programm das Environment abfragen zu können, bekommt unter den meisten Systemen die Funktion `main()` einen dritten Parameter. Dieser ist ein Zeiger auf ein Feld von nullterminierten Strings (ähnlich wie der zweite Parameter auf die Programmargumente zeigt), also vom Typ `char **` beziehungsweise `char *[]`. Jeder dieser Strings enthält den Namen einer Variablen, dann ein Gleichheitszeichen (`=`), gefolgt vom Wert der Variable. Ein Beispiel für einen solchen String könnte `"HOME=/home/klaus"` sein; darin wird die Variable `HOME` mit dem Wert `"/home/klaus"` definiert.

Die Anzahl der vorhandenen Variablen wird nicht übergeben (anders als bei den Programmargumenten); stattdessen ist hinter der letzten Variable im Feld von `char*`-Zeigern eine `NULL` abgelegt, daran muß man beim Abfragen das Ende erkennen.

Dieser Mechanismus ist nicht im ANSI-Standard enthalten, aber sehr weit verbreitet (alle unixähnlichen Systeme, unter Windows zumindest wenn man mit `gcc`, Visual C++, Borland oder Visual Age arbeitet)

Das folgende Programm zeigt, wie man in `main()` an die Parameter kommt:

```
/* Time-stamp: "03.10.02 08:19 pamainenv.c klaus@wachtler.de"
 *
 * (Ausgabe aller envirnoment-Variablen; kein ANSI-Standard!)
 */

int main( int n, char **s, char **env_pp )
{
  /* Schleife über alle Environmentvariablen
   * (nach der letzten ist noch ein NULL-Zeiger im
   * Feld, woran man das Ende erkennen kann):
   */
  while( *env_pp )
  {
    /* *env_pp ist noch nicht NULL, also ausgeben:
     */
    puts( *env_pp );

    env_pp++;  /* Zeiger auf die nächste Env.-Var. setzen */
  }

  return 0;
}
```

Die (gekürzte) Ausgabe könnte so aussehen:

```
PWD=/home/klaus/skript_c
MAKEFLAGS=
WINDOWID=8388622
PAGER=less
...
```

Hinweis: Mit **getenv()** (Seite 199) gibt es einen portablen Weg, den Wert einer Environmentvariablen zu erfahren, falls man ihren Namen kennt (dies ist überall im Programm möglich, nicht nur in `main()`).

Eine portable Möglichkeit, alle Environmentvariablen zu erfragen, ohne vorher ihre Namen zu kennen, existiert nicht.

15 Dateien

Die Verwendung von Dateien auf Massenspeichern ist im eigentlichen Sprachkonzept von *C* überhaupt nicht vorgesehen; vielmehr wird die gesamte Anbindung an die Welt außerhalb des Arbeitsspeichers des Rechners über Aufrufe von Funktionen realisiert.

In allen älteren *C*-Versionen (*K&R*) gibt es dabei zwei verschiedene Dateikonzepte:

- relativ primitive Funktionen zur direkten Verwendung der Unix-Funktionen. Diese sogenannte *ungepufferte Ein-/Ausgabe* oder *low level IO* bestand zumindest im Original unter Unix weitgehend aus Makrodefinitionen in der Datei io.h. Auf anderen Systemen wurden diese Funktionen mehr oder weniger genau nachgebildet. Der Umfang der Funktionen beschränkte sich im wesentlichen auf das Öffnen und Schließen von Dateien, Lesen und Schreiben von einzelnen Zeichen und Speicherblöcken und Manipulieren der Dateiattribute, also relativ primitive Funktionen.

  Programme, die diese Funktionen verwenden, sind schwer oder gar nicht portierbar.

  Die Funktionen existieren nicht mehr im (neueren) *ANSI-C*. Die meisten *C*-Systeme bieten sie zwar noch an, aber für Neuentwicklungen sollte man die Finger davon lassen[29].

  In diesem Buch werden sie nicht weiter behandelt.

- die sogenannte *gepufferte* Ein- und Ausgabe.

  Die dazugehörigen Funktionen sind in der Headerdatei stdio.h deklariert.

  Damit kann man wesentlich komfortabler und systemunabhängiger Programme schreiben, da die Funktionen sehr leistungsfähig sind und außerdem gut genormt.

  Die Dateifunktionen greifen dabei in der Regel nicht für jeden Lese- oder Schreibzugriff auf die Platte zu, sondern halten in einem Puffer den gerade benötigten Bereich. Erst bei Überschreiten dieses Puffers wird wieder ein neuer Bereich gelesen oder geschrieben. Dadurch werden die Zugriffe auf die meist langsamen Massenspeicher minimiert und damit im Schnitt schneller.

Im weiteren wird nur noch die *gepufferte Ein- und Ausgabe* verfolgt.

Die Funktionen werden hier nicht detailliert vorgestellt, sondern nur das eigentliche Konzept erklärt.

In der Datei stdio.h ist ein Datentyp `FILE` vereinbart (mittels `typedef`). `FILE` ist eine Struktur und enthält alle Informationen, welche die Bibliotheksfunktionen zu einer geöffneten Datei verwalten müssen, wie etwa Dateiname, Kennummer der Datei für das Betriebssystem (*file handle*), Lage, Größe und Gültigkeit des aktuellen Puffers, eventuell aufgetretene Fehler und so weiter. Diese Informationen brauchen den Benutzer nicht zu interessieren; es ist nicht legal, auf die einzelnen Elemente einer solchen Struktur zuzugreifen. Selbst wenn der Zugriff formal möglich ist, kann man ein solches Programm nicht mehr portieren.

Neben der Struktur `FILE` existiert noch ein spezieller Datentyp für die Position in Dateien, nämlich `fpos_t`. Dies ist ein ganzzahliger Datentyp (meist `unsigned long`

[29] Zumindest solange die Programme portierbar bleiben sollen. Für systemnahe Probleme sind sie nach wie vor noch manchmal sinnvoll, aber dann nicht mehr rechnerunabhängig.

int bzw. size_t), der eine Dateiposition speichern kann. Dieser Datentyp wird nur für fgetpos() und fsetpos() benötigt.

Die gesamte Ein- und Ausgabe basiert auf der Vorstellung von Datenströmen (*stream*, also auf irgendwelchen dunklen Kanälen, in denen die Daten im Gänsemarsch entlangdüsen. Dabei hat jede offene Datei einen solchen Kanal zur Verfügung.

Mit Strom ist dabei gemeint, daß in der Regel alle Daten einer Datei oder eines Geräts hintereinander geschrieben oder gelesen werden. Man kann sich also einen Zeiger in die Datei denken, der nach dem Öffnen meist auf dem ersten Zeichen der Datei steht und bei jeder Schreib- oder Leseoperation über die transferierten Zeichen hinwegbewegt wird, so daß die nächste Operation hinter der letzten stattfindet. Es gibt aber Funktionen, welche die normale Reihenfolge (*sequentiell*) aufbrechen und diesen gedachten Zeiger frei in der Datei positionieren lassen (*wahlfreier Zugriff*). Dies sind insbesondere die Funktionen fsetpos(), fseek() und rewind().

Ein solcher Kanal ist dabei aber nicht an eine Datei auf Diskette oder Magnetplatte gebunden; ebenfalls mit Kanälen werden nämlich die Ausgabe auf den Bildschirm, das Lesen von der Tastatur, die Druckerausgabe und eventuell anderes abgewickelt.

Will man einen Datenstrom verwenden, muß man ihn in der Regel mit der Funktion fopen() öffnen. Dabei erhält man einen Zeiger auf eine Struktur vom Typ FILE zurück. Diesen Zeiger braucht man nur beim Aufruf anderer Funktionen aus stdio.h[30]. fopen() erhält zwei Parameter. Der erste der beiden ist ein String, der den Namen der zu öffnenden Datei enthält. Je nach Rechnersystem sind die Regeln zur Bildung von Dateinamen unterschiedlich.

Der zweite Parameter (Dateimodus) ist ebenfalls ein String, der angibt, wie die Datei zu öffnen ist. Mögliche Werte sind"r", "w", "a", "rb", "wb", "ab", "r+", "w+", "a+", "r+b", "w+b", "a+b", "rb+", "wb+" und "ab+". Ein r im zweiten Parameter besagt, daß die Datei nur gelesen werden soll und deshalb schon existieren muß. Mit einem w gibt man an, daß die Datei nur beschrieben werden soll. Existiert sie schon, dann wird sie überschrieben, also der alte Inhalt gelöscht. Ein a ist wie w, aber die Datei wird anfangs nicht auf den Dateianfang, sondern auf das Dateiende positioniert und nicht gelöscht, wenn sie schon existiert. Mit einem r+ sagt man, daß die Datei beschrieben und gelesen werden soll (*update*). w+ ist fast identisch; aber wenn die Datei schon existiert, dann wird der alte Inhalt gelöscht. a+ ist wie r+, aber der Dateizeiger steht anfangs am Ende.

Ein b besagt, daß die Datei als Binärdatei behandelt werden soll. Ansonsten ist es eine Textdatei. Zum Unterschied siehe **fopen()** (Seite 192).

Wenn eine Datei zum Schreiben *und* Lesen geöffnet wird, dann muß jeder Wechsel zwischen Lesen und Schreiben oder umgekehrt von fflush() oder einer Dateipositionierung begleitet werden, wie etwa fseek() oder rewind(). Sonst kann die Pufferung nämlich nicht klappen.

Für die am häufigsten benötigten Ströme sind allerdings bei Programmstart schon Kanäle geöffnet: stdin zum Lesen von der Standardeingabe (das ist in der Regel die Tastatur), stdout zum Schreiben auf die Standardausgabe (in der Regel der Bildschirm) und stderr als Kanal für Fehlermeldungen (meistens ebenfalls der Bildschirm). Manche Systeme bieten weitere bei Programmstart geöffnete Kanäle an, wie stdprn zum Schreiben auf den Drucker und stdaux als zusätzlicher Kanal, der meistens mit einer seriellen Schnittstelle wie einem Modem verbunden ist; diese sind aber nicht standardisiert. Die vorgöffneten Dateien std... sind globale Variablen vom Typ FILE *.

[30] Anders als zur Übergabe an Funktionen sollte dieser Zeiger nicht verwendet werden; insbesondere darf man die Struktur, auf die er zeigt, nicht manipulieren!

Nur `stdin`, `stdout` und `stderr` sind im ANSI-Standard definiert.

Will man einen Strom, also einen Ein- oder Ausgabekanal, öffnen, der noch nicht vordefiniert ist, dann muß man eine Variable vom Typ `FILE *` vereinbaren und `fopen()` aufrufen. Der Rückgabewert von `fopen()` wird der Variablen zugewiesen. Wenn die Datei aus irgendwelchen Gründen nicht wunschgemäß geöffnet werden kann, dann liefert `fopen()` den Wert `NULL` zurück; ansonsten einen Zeiger auf eine Variable vom Typ `FILE`. Diesen Zeiger muß man an eine eigene Variable vom Typ `FILE *` zuweisen und für die nachfolgenden Aufrufe der Dateifunktionen aufbewahren. (Mehr als Aufbewahren und an andere Funktionen übergeben darf man damit auch nicht machen.)

Auf vielen Systemen kann man nur wenige Dateien gleichzeitig geöffnet halten (meist etwa 10 oder 20). Etwas Bescheidenheit ist hier von Nutzen.

Alle weiteren Funktionen beziehen sich auf einen solchen Zeiger, um auf die geöffnete Datei zugreifen zu können.

`fclose()` schließt eine Datei und ist damit das Gegenstück zu `fopen()`. Nach dem Aufruf von `fclose()` darf auf die Datei nicht mehr zugegriffen werden. Beim Programmende werden alle noch offenen Dateien automatisch geschlossen; aber da die Anzahl der gleichzeitig offenen Dateien meist begrenzt ist, sollte man Dateien schließen, wenn sie nicht mehr benötigt werden.

`fgetc()` liest ein Zeichen von einem Datenstrom, `fputc()` gibt ein Zeichen auf einen Datenstrom aus.

`fread()` und `fwrite()` lesen und schreiben gleich einen ganzen Block in den Speicher oder in die Datei.

`fgets()` und `fputs()` lesen beziehungsweise schreiben einen String.

`fprintf()` und `fscanf()` führen formatierte Ein- und Ausgabe durch, wie `printf()` und `scanf()`, nur eben mit Dateien anstatt der Standardein- und -ausgabe.

Die ansonsten mit `fprintf()` und `fscanf()` identischen Funktionen `sprintf()` und `sscanf()` übertragen nicht nach und von Dateien, sondern in einen String oder lesen von dort. Sie haben außer der Ähnlichkeit mit `printf()` beziehungsweise `scanf()` nichts mit Dateien zu tun, sind aber trotzdem in stdio.h vereinbart.

Mit den Paaren `ftell()` und `fseek()` beziehungsweise `fgetpos()` und `fsetpos()` kann man die aktuelle Dateiposition erfragen und ändern.

`rewind()` positioniert eine Datei wieder an den Anfang.

Mit `setbuf()` und `setvbuf()` kann man auf die Pufferung Einfluß nehmen.

`feof()` prüft, ob ein Dateizeiger schon am Ende der Datei angelangt ist; `ferror()` prüft, ob der Fehlerindikator für die betreffende Datei gesetzt ist.

`clearerr()` löscht einen eventuell gesetzten Fehlerindikator.

Einige Funktionen liefern im Fehlerfall den Wert `EOF` (üblicherweise als `-1` in stdio.h definiert).

Für einige Funktionen existieren Varianten für die Standardeingabe `stdin` oder die Standardausgabe `stdout`. So ist `printf( ... )` identisch mit `fprintf( stdout, ... )`.

Das folgende kleine Beispiel schreibt einige `int`-Werte in eine Textdatei (und zwar einen Wert je Zeile) und liest die Werte nach dem Zurückspulen wieder aus:

```
/* sttxt.c 30. 7.95 kw
 */

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

int main()
{
  FILE   *f;
  int     wert;

  /* Die Datei oeffnen:
   */
  f = fopen( "sttxt.dat", "w+" );
  if( f==NULL )  /* Fehler? */
  {
    printf( "Ich kann die Datei nicht oeffnen!\n" );
    return EXIT_FAILURE;
  }

  /* Einige Werte schreiben:
   */
  fprintf( f, "%d\n%d\n%d\n", 123, 456, 789 );
  fprintf( f, "%d\n%d\n%d\n", 14, 25, 36 );

  /* Die Datei an den Anfang zurueckspulen:
   */
  rewind( f );

  /* Die Werte lesen, bis Dateiende auftritt:
   */
  while( fscanf( f, "%d", &wert )==1 )
  {
    /* Kontrollausgabe auf den Bildschirm:
     */
    printf( "gelesener Wert: %d\n", wert );
  }

  /* Datei wieder schliessen:
   */
  fclose( f );

  /* Programm beenden:
   */
  return EXIT_SUCCESS;
}
```

Nach dem Lauf dieses Programms existiert eine Textdatei mit dem Namen sttxt.dat und folgendem Inhalt:

```
123
456
789
```

```
14
25
36
```

Will man keine Textdatei haben, dann ist eine entsprechende unformatierte Binärdatei meist platzsparender und vor allem schneller zu schreiben und lesen (vom Programm aus), da die Umwandlung der programminternen Binärdarstellung in Textzeichen entfällt.

Das Beispielprogramm würde dann so aussehen:

```
/* stbin.c 30. 7.95 kw
 */

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

int main()
{
  FILE   *f;
  int     wert;

  /* Die Datei oeffnen:
   */
  f = fopen( "test.dat", "wb+" ); /* b==Binaerdatei!    */
  if( NULL==f )                   /* Fehler?            */
  {
    printf( "Ich kann die Datei nicht oeffnen!\n" );
    return EXIT_FAILURE;
  }

  /* Einige Werte schreiben:
   */
  wert = 123;
  fwrite( &wert, sizeof(wert), 1, f );
  wert = 456;
  fwrite( &wert, sizeof(wert), 1, f );
  wert = 789;
  fwrite( &wert, sizeof(wert), 1, f );
  wert = 14;
  fwrite( &wert, sizeof(wert), 1, f );
  wert = 25;
  fwrite( &wert, sizeof(wert), 1, f );
  wert = 36;
  fwrite( &wert, sizeof(wert), 1, f );

  /* Die Datei an den Anfang zurueckspulen:
   */
  rewind( f );

  /* Die Werte lesen, bis Dateiende auftritt:
   */
  while( fread( &wert, sizeof(wert), 1, f )==1 )
  {
```

```
    /* Kontrollausgabe auf den Bildschirm:
     */
    printf( "gelesener Wert: %d\n", wert );
  }

  /* Datei wieder schliessen:
   */
  fclose( f );

  /* Programm beenden:
   */
  return EXIT_SUCCESS;
}
```

16 Aufteilung auf mehrere Quelltexte

Ab einer gewissen Programmgröße ist es sinnvoll, den Quelltext auf mehrere Dateien aufzuteilen. Dies ist mit mehreren Vorteilen verbunden:

- Die Übersetzungszeiten reduzieren sich, da bei kleineren Änderungen nicht das gesamte Programm übersetzt werden muß, sondern nur die geänderten Dateien.
- Mit einer thematisch sinnvollen Aufteilung steigt die Übersichtlichkeit und Wartbarkeit des Programms deutlich.

  So sollten alle systemabhängigen Teile (elementare Grafikausgabe, maschinenabhängiges Ansprechen von AD-Wandlern, usw.) in eigenen Dateien untergebracht werden. Die eigentliche Programmlogik selbst kann in portablem *C* in anderen Dateien stehen.
- Besondere Datentypen können zusammen mit den Funktionen, die solche Daten manipulieren, in eigenen Dateien stehen (Kapselung).
- Die Fehlersuche vereinfacht sich enorm, wenn zuverlässig arbeitende Hilfsfunktionen separat gehalten werden von den Programmteilen, die noch entwickelt werden und fehlerhaft arbeiten.
- Die Arbeitsteilung bei größeren Projekten ist wesentlich einfacher, wenn jeder Mitarbeiter unabhängig von anderen an seinen Funktionen arbeiten kann.

Auf der anderen Seite handelt man sich beim Aufteilen auf mehrere Quelltexte etliche neue Fehlerquellen ein, da der *C*-Compiler immer nur einen Quelltext auf einmal „sieht“ und über Quelldateien hinweg keine Fehlerüberprüfung machen kann. Abgesehen von Hilfsprogrammen dazu (`lint` unter *Unix*) kann man durch Einhaltung einer gewissen Systematik die Gefahren aber relativ leicht und effektiv umschiffen. So ist dieser Abschnitt weitgehend als Empfehlung zu verstehen (aber als dringende!).

Die allgemein nötige Verwaltungsarbeit der Modularisierung (klare Definition der Schnittstellen, Dokumentation der *black boxes*, etc. sei hier vorausgesetzt; ich will nur die für *C* spezifischen Punkte aufführen.

C unterstützt generell die getrennte Kompilierung eines Programms in mehreren Teilen. Die kleinste Einheit, die unteilbar in einem Quelltext stehen muß, ist für Daten eine Variable (einfache, Struktur oder Feld) und für Programmcode eine vollständige Funktion. Das heißt, daß beispielsweise ein Feld nicht teilweise in einer Quelldatei und teilweise in einer anderen definiert werden kann.

Alle in einer Quelldatei definierten globalen Objekte (Variablen und Funktionen), die nicht als `static` deklariert sind, können von allen anderen Quelltexten aus als `extern` deklariert und angesprochen werden. Die Definition eines jeden Objektes darf allerdings in allen Quelltexten nur einmal erfolgen.

Dieser Mechanismus unterscheidet sich also von *FORTRAN*: Dort werden die globalen Daten in allen Quelltexten gleich vereinbart, nämlich als `COMMON`.

In *C* dagegen muß man dafür sorgen, daß alle Objekte in allen Dateien gleich deklariert werden, außer in einer Datei, die das Objekt auch definiert[31].

In welchen Quelldateien man seine globalen Variablen definiert, ist formal gleichgültig. Nach Belieben kann man auch separate Quelltexte schreiben, in denen nur Datendefinitionen stehen; ein *C*-Quelltext muß keine Funktionen enthalten.

[31] Zum Unterschied zwischen *Definition* und *Deklaration* siehe Abschnitt 3 auf Seite 14.

Um den Mechanismus zu verdeutlichen, soll ein kleines Projekt vorgestellt werden:

- Das Programm fragt nach dem Benutzernamen und merkt ihn sich.
- Eine Maßeinheit wird eingelesen und ebenfalls gemerkt.
- Solange der Benutzer positive Radien eingibt, werden zu diesem Radius Umfang und Fläche des zugehörigen Kreises berechnet und ausgegeben. Die Ausgabe erfolgt auf den Bildschirm und in eine Protokolldatei.

  Bei Eingabe einer nicht positiven Zahl wird das Programm beendet.

Dieses Projekt läßt sich natürlich auch leicht in einem Quelltext realisieren:

```
/* sp1q.c 30. 7.95 kw
 */

#include <stdio.h>

#define     PI      3.14159265  /* Kreiszahl                    */
#define     LENG    20          /* solange darf der Name sein   */

char        name[LENG];
char        masseinheit[LENG];

FILE       *log_f;

/* Holt eine double von der Tastatur:
 */
double hole_double( const char *s1 )
{
  double wert;

  printf( "Bitte %s [%s] eingeben, lieber %s: ",
          s1,         masseinheit,
          name
          );
  scanf( "%lf", &wert );
  return wert;
}

/* Holt einen String:
 */
void hole_string( const char *s, char *wohin )
{
  printf( "Bitte %s eingeben: ", s );
  scanf( "%s", wohin );
}

/* gibt einen Radius in die Datei log_f aus:
 */
void schreibe_radius( double radius )
{
  fprintf( log_f, "Radius %f [%s]\n", radius, masseinheit );
}
```

```c
void schreibe_umfang_flaeche( double umfang, double flaeche )
{
  printf( "\nUmfang = %f [%s], Flaeche = %f [%s*%s]\n",
          umfang,
          masseinheit,
          flaeche,
          masseinheit,
          masseinheit
          );
  fprintf( log_f,
           "\nUmfang = %f [%s], Flaeche = %f [%s*%s]\n",
           umfang,
           masseinheit,
           flaeche,
           masseinheit,
           masseinheit
           );
}

/* Macht alle Initialisierungen:
 */
void start( void )
{
  /* Die Protokolldatei oeffnen:
   */
  if( (log_f=fopen( "RADIUS.LOG", "w" ))==NULL )
  {
    printf( "Ich kann die "
            "Protokolldatei nicht oeffnen!\n" );
    exit( 3 );
  }

  /* Den Benutzernamen holen:
     */
  hole_string( "Ihren Namen", name );

  /* dto. die Einheit:
     */
  hole_string( "die Masseinheit", masseinheit );

  /* Den Kopf der Protokolldatei schreiben:
     */
  fprintf( log_f,
           "Protokolldatei Kreisberechnung\n"
           "Benutzer %s, Masseinheit [%s]\n",
           name,
           masseinheit
           );
}

/* Raeumt alles wieder auf:
 */
void ende( int rueckgabe )
```

```
{
  fprintf( log_f, "\nProgrammende!\n" );
  fclose( log_f );
  printf( "\nSchoenen Tag noch, %s!\n", name );
  exit( rueckgabe );
}

/* Berechnet zu einem Radius den Kreisumfang:
 */
double rechne_umfang( double r )
{
  return 2.0*r*PI;
}

/* Berechnet zu einem Radius die Kreisflaeche:
 */
double rechne_flaeche( double r )
{
  return r*r*PI;
}

int main()
{
  double
    radius,
    umfang,
    flaeche;

  start();   /* Datei Oeffnen etc.                         */

  /* Solange Radius positiv: Lesen ...
   */
  while(   (radius=hole_double( "Radius" ))
           > 0.0
           )
  {
    /* ... Rechnen ...
     */
    umfang  = rechne_umfang( radius );
    flaeche = rechne_flaeche( radius );
    /* ... und Ausgeben:
     */
    schreibe_radius( radius );
    schreibe_umfang_flaeche( umfang, flaeche );
  }

  /* alles aufraeumen und Programm beenden.
   */
  ende( 0 );
}
```

Globale Objekte in diesem Programm sind die Variablen `name`, `masseinheit` und `log_f` sowie alle Funktionsnamen.

Das Programm soll nun sinnvoll auf kleinere Quelldateien aufgeteilt werden, da es dem hochgeschätzten Vorgesetzten zu groß ist und er schon mal etwas von modularer Programmierung gehört hat. Dazu muß man die globalen Variablen in jeder Datei als `extern` deklarieren, außer in einer. Dort muß die jeweilige Variable wie sonst auch definiert werden.

Wir teilen das Programm in

- ein Modul zur Ein- und Ausgabe (Quelldatei spio1.c),
- ein Modul mit allen rechnenden Funktionen (sprech1.c),
- ein Modul mit dem Hauptprogramm, `start()`, `ende()` und den Definitionen der globalen Variablen (Quelldatei sphaupt1.c).

Die aufgerufenen Funktionen sollten vor ihrer ersten Verwendung in jeder Quelldatei deklariert werden. Dann sieht das erste Modul (Ein-/Ausgabe) so aus:

```
/* spio1.c 30. 7.95 kw
 */

#include <stdio.h>

#define     PI      3.14159265  /* Kreiszahl                  */
#define     LENG    20          /* solange darf der Name sein   */

extern char     name[LENG];
extern char     masseinheit[LENG];

extern FILE    *log_f;

/* Holt eine double von der Tastatur:
 */
double hole_double( const char *s1 )
{
  double wert;

  printf( "Bitte %s [%s] eingeben, lieber %s: ",
          s1,
          masseinheit,
          name
          );
  scanf( "%lf", &wert );
  return wert;
}

/* Holt einen String:
 */
void hole_string( const char *s, char *wohin )
{
  printf( "Bitte %s eingeben: ", s );
  scanf( "%s", wohin );
}

/* gibt einen Radius in die Datei log_f aus:
```

```
 */
void schreibe_radius( double radius )
{
  fprintf( log_f, "Radius %f [%s]\n", radius, masseinheit );
}

void schreibe_umfang_flaeche( double umfang, double flaeche )
{
  printf( "\nUmfang = %f [%s], Flaeche = %f [%s*%s]\n",
          umfang,
          masseinheit,
          flaeche,
          masseinheit,
          masseinheit
          );
  fprintf( log_f,
           "\nUmfang = %f [%s], Flaeche = %f [%s*%s]\n",
           umfang,
           masseinheit,
           flaeche,
           masseinheit,
           masseinheit
           );
}
```

Das Modul sprech1.c könnte so aussehen:

```
/* sprech1.c 30. 7.95 kw
 */

#define     PI      3.14159265  /* Kreiszahl                     */
#define     LENG    20          /* solange darf der Name sein    */

extern char     name[LENG];
extern char     masseinheit[LENG];

/* Berechnet zu einem Radius den Kreisumfang:
 */
double rechne_umfang( double r )
{
  return 2.0*r*PI;
}

/* Berechnet zu einem Radius die Kreisflaeche:
 */
double rechne_flaeche( double r )
{
  return r*r*PI;
}
```

Das Hauptmodul enthält von den aufgerufenen Funktionen nur noch die Deklarationen (Funktionsprototypen). Die globalen Variablen sind hier dagegen definiert, während sie in den beiden obigen Quelltexten als `extern` deklariert sind:

```
/* sphaupt1.c 30. 7.95 kw
 */

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

#define     PI      3.14159265  /* Kreiszahl                    */
#define     LENG    20          /* solange darf der Name sein   */

/* Die Variablen hier nicht als extern deklarieren, sondern
 * definieren!!!
 */

char        name[LENG];
char        masseinheit[LENG];

FILE       *log_f;

/* Die aufgerufenen Funktionen deklarieren:
 */
double hole_double( const char *s1 );
void hole_string( const char *s, char *wohin );
void schreibe_radius( double radius );
void schreibe_umfang_flaeche( double umfang, double flaeche );
double rechne_umfang( double r );
double rechne_flaeche( double r );

/* Macht alle Initialisierungen:
 */
void start( void )
{
  /* Die Protokolldatei oeffnen:                              */
  if( (log_f=fopen( "RADIUS.LOG", "w" ))==NULL )
  {
    printf( "Ich kann die "
            "Protokolldatei nicht oeffnen!\n" );
    exit( 3 );
  }

  /* Den Benutzernamen holen:
     */
  hole_string( "Ihren Namen", name );

  /* dto. die Einheit:
     */
  hole_string( "die Masseinheit", masseinheit );

  /* Den Kopf der Protokolldatei schreiben:
     */
  fprintf( log_f,
           "Protokolldatei Kreisberechnung\n"
           "Benutzer %s, Masseinheit [%s]\n",
           name,
           masseinheit                       );
```

```
}

/* Raeumt alles wieder auf:
 */
void ende( int rueckgabe )
{
  fprintf( log_f, "\nProgrammende!\n" );
  fclose( log_f );
  printf( "\nSchoenen Tag noch, %s!\n", name );
  exit( rueckgabe );
}

int main()
{
  double
    radius,
    umfang,
    flaeche;

  /* Datei oeffnen etc.
   */
  start();

  /* Solange Radius positiv: Lesen ...
   */
  while(   (radius=hole_double( "Radius" ))
           > 0.0
           )
  {
    /* ... Rechnen ...
     */
    umfang  = rechne_umfang( radius );
    flaeche = rechne_flaeche( radius );
    /* ... und Ausgeben:
     */
    schreibe_radius( radius );
    schreibe_umfang_flaeche( umfang, flaeche );
  }

  /* alles aufrauemen und Programm beenden.
   */
  ende( 0 );
}
```

Nach dem getrennten Kompilieren dieser drei Quelltexte werden die erzeugten Objektdateien mit einem Linker zu einem lauffähigen Programm zusammengebunden.

Diese Vorgehensweise funktioniert zwar, hat aber einige Nachteile:

- Die Aufteilung auf die Quelltexte erfordert viel Handarbeit und eine klare Vorstellung vom Zusammenspiel der Module.
- Viele Vereinbarungen kommen mehrfach vor. Alle Deklarationen und Definitionen stehen (fast oder völlig) gleich in mehreren Dateien.

Das ist sowohl beim Aufteilen selbst als auch bei nachträglichen Änderung aufwendig und fehleranfällig.

Diese Nachteile lassen sich durch gezielten Einsatz des Praeprozessors eliminieren. Zuerst kann man die identischen Teile der Quelltexte in einer sogenannten *header*-Datei zusammenfassen und mit `#include` in alle Module einbinden. Wenn ein Objekt (Variable oder Funktion) vor seiner Definition schon deklariert wird, dann ist das kein Fehler. So kann man die Headerdatei mit den Deklarationen auch in den Modulen `#include`n, in denen die Objekte danach noch definiert werden.

Die Vereinbarungen:

```
/* Deklarationen:               */
extern int variable;
void funktion( int i );

/* Definitionen:                */
int variable;
void funktion( int i )
    {   printf( "%d", i );
    }
```

sind also zulässig.

Die Headerdatei spkreis.h in unserem Beispiel könnte also so aussehen:

```
/* spkreis.h 30. 7.95 kw
 */

#define     PI      3.14159265  /* Kreiszahl                    */
#define     LENG    20          /* solange darf der Name sein   */

/* globale Variablen:                                           */
extern char        name[LENG];
extern char        masseinheit[LENG];

extern FILE       *log_f;

/* Die aufgerufenen Funktionen deklarieren:
 */
double hole_double( const char *s1 );
void hole_string( const char *s, char *wohin );
void schreibe_radius( double radius );
void schreibe_umfang_flaeche( double umfang, double flaeche );
double rechne_umfang( double r );
double rechne_flaeche( double r );
void start( void );
void ende( int rueckgabe );
```

Diese Datei kann in alle Quelldateien mit #include eingebunden werden. Eigene Vereinbarungen werden üblicherweise im gleichen Verzeichnis bzw. *directory* stehen wie die Quelltexte; dann muß man die Form #include "..." verwenden. Nur die Includedateien, welche bei den Standardheaderdateien (z.B. stdio.h) zu finden sind, werden mit #include <...> eingefügt.

Die Namen von Headerdateien enden fast immer auf .h, aber das ist nur Konvention. Die Dateinamen für eigene Headerdateien sind (je nach Betriebssystem) mehr oder weniger frei wählbar.

Die Quelldateien spio2.c, sprech2.c und sphaupt2.c sehen nun so aus (die einzelnen Dateien sind mit einem Strich voneinander abgesetzt):

```
/* spio2.c 30. 7.95 kw
 */

#include <stdio.h>

#include "kreis.h"

/* Holt eine double von der Tastatur:
 */
double hole_double( const char *s1 )
{
  double wert;

  printf( "Bitte %s [%s] eingeben, lieber %s: ",
          s1,
          masseinheit,
          name
          );
  scanf( "%lf", &wert );
  return wert;
}

/* Holt einen String:
 */
void hole_string( const char *s, char *wohin )
{
  printf( "Bitte %s eingeben: ", s );
  scanf( "%s", wohin );
}

/* gibt einen Radius in die Datei log_f aus:
 */
void schreibe_radius( double radius )
{
  fprintf( log_f, "Radius %f [%s]\n", radius, masseinheit );
}

void schreibe_umfang_flaeche( double umfang, double flaeche )
{
  printf( "\nUmfang = %f [%s], Flaeche = %f [%s*%s]\n",
          umfang,
          masseinheit,
```

```
          flaeche,
          masseinheit,
          masseinheit
          );
  fprintf( log_f,
           "\nUmfang = %f [%s], Flaeche = %f [%s*%s]\n",
           umfang,
           masseinheit,
           flaeche,
           masseinheit,
           masseinheit
           );
}
```

```
/* sprech2.c 30. 7.95 kw
 */

#include <stdio.h>

#include "kreis.h"

/* Berechnet zu einem Radius den Kreisumfang:
 */
double rechne_umfang( double r )
{
  return 2.0*r*PI;
}

/* Berechnet zu einem Radius die Kreisflaeche:
 */
double rechne_flaeche( double r )
{
  return r*r*PI;
}
```

```
/* sphaupt2.c 30. 7.95 kw
 */

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

#include "kreis.h"

/* globale Variablen definieren:
 */
char        name[LENG];
char        masseinheit[LENG];

FILE       *log_f;
```

```
/* Macht alle Initialisierungen:
 */
void start( void )
{
  /* Die Protokolldatei oeffnen:
   */
  if( (log_f=fopen( "RADIUS.LOG", "w" ))==NULL )
  {
    printf( "Ich kann die "
            "Protokolldatei nicht oeffnen!\n"
            );
    exit( 3 );
  }

  /* Den Benutzernamen holen:
     */
  hole_string( "Ihren Namen", name );

  /* dto. die Einheit:
     */
  hole_string( "die Masseinheit", masseinheit );

  /* Den Kopf der Protokolldatei schreiben:
   */
  fprintf( log_f,
           "Protokolldatei Kreisberechnung\n"
           "Benutzer %s, Masseinheit [%s]\n",
           name,
           masseinheit
           );
}

/* Raeumt alles wieder auf:
 */
void ende( int rueckgabe )
{
  fprintf( log_f, "\nProgrammende!\n" );
  fclose( log_f );
  printf( "\nSchoenen Tag noch, %s!\n", name );
  exit( rueckgabe );
}

int main()
{
  double
    radius,
    umfang,
    flaeche;

  /* Datei oeffnen etc.
   */
  start();
```

```
  /* Solange Radius positiv: Lesen ...
   */
  while(   (radius=hole_double( "Radius" ))
           > 0.0
           )
  {
    /* ... Rechnen ...
     */
    umfang  = rechne_umfang( radius );
    flaeche = rechne_flaeche( radius );
    /* ... und Ausgeben:
     */
    schreibe_radius( radius );
    schreibe_umfang_flaeche( umfang, flaeche );
  }

  /* alles aufrauemen und Programm beenden.
   */
  ende( 0 );
}
```

Damit ist die Redundanz deutlich geringer geworden: Alle Vereinbarungen kommen nur noch an zwei Stellen vor, nämlich in der Headerdatei als Deklaration und im jeweiligen Quelltext als Definition. Durch den Praeprozessor definierte Konstanten (z.B. `PI`) stehen nur in kreis.h.

Zumindest bei Datenvereinbarungen läßt sich aber auch dieses doppelte Vorkommen noch mit Hilfe des Praeprozessors und einem kleinen Trick beseitigen. Der einzige Unterschied zwischen einer Deklaration und der Definition liegt in der Verwendung des Schlüsselworts `extern`. Dieses wird vom Compiler ausgewertet. Überredet man den Praeprozessor jetzt dazu, in einer Datei alle `extern`-Schlüsselworte zu entfernen, dann werden die Variablen allein durch Einfügen der Headerdatei auch definiert.

Den Anfang von sphaupt2.c ändert man also zu:

```
/* sphaupt3.c
 */

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

#define extern
#include "kreis.h"
#undef extern

/* globale Variablen brauchen jetzt nicht mehr definiert zu
 * werden!
 */


/* Macht alle Initialisierungen:
 */
void start( void )
    {
        ...
```

Zur Erklärung: Durch `#define extern` werden alle folgenden Vorkommen von `extern` durch nichts ersetzt, auch während kreis.h eingefügt wird. Das nachfolgende `#undef extern` hebt das `#define` wieder auf (für den Fall, daß im weiteren Quelltext doch noch `extern` benötigt wird).

Ein Problem taucht auf, wenn globale Variablen gleich mit der Vereinbarung initialisiert werden sollen. Dies darf nur bei der Definition geschehen; bei der reinen Deklaration ist die Initialisierung verboten. Das Problem kann man aber auch mit dem Praeprozessor lösen, nämlich mit bedingter Kompilierung.

Beispiel: Eine globale Variable `i` soll in einer Headerdatei vereinbart werden und gleich mit dem Wert `10` initialisiert werden.

Dazu schreibt man in die Headerdatei:

```
extern int      i
#ifdef extern
                   = 10
#endif
                       ;
```

Dann kann man (wie oben beschrieben) mit

```
    #define extern
    #include "..."
    #undef extern
```

die Datei zur Definition (und Initialisierung) einfügen, dagegen nur mit

```
    #include "..."
```

zur Deklaration.

Durch diese Vorgehensweise kommen alle Vereinbarungen nur noch einfach (Variablen, Praeprozessorkonstanten) oder zweifach (Funktionsprototypen) vor.

Dateilokale Objekte: Soll eine Funktion oder eine globale Variable nur von der Datei aus verwendet werden können, in der sie auch definiert ist (also eben nicht global bezüglich aller Quelltexte), dann muß man sie als `static` vereinbaren (siehe **8.6 Attributangaben** auf Seite 50). Damit wird der Name des Objekts nicht dem Linker bekannt gemacht wie sonst alle globalen Namen eines Quelltextes, sondern ist nur in der aktuellen Datei ansprechbar (*information hiding*).

Mehrfaches `#include`n: Manchmal kommt es vor, daß eine Includedatei versehentlich mehrfach eingefügt wird, beispielsweise durch geschachtelte `#include`-Anweisungen. Deshalb wird üblicherweise jede Includedatei von einer `#ifndef`-Anweisung eingeschlossen und das zugehörige Makro dazwischen definiert.

Beispiel: Die Datei meine.h soll so aussehen:

```
/* include-Datei meine.h vom 16. 8.91                          */
#define PI    3.141592654          /* Kreiszahl                */
#define EU    2.718281828          /* Eulersche Zahl           */
int wieoft;                        /* eine globale Variable    */
```

Wird diese Datei versehentlich mehrfach eingefügt, dann sind die Definitionen für den Praeprozessor und den Compiler entsprechend mehrfach vorhanden. Die Lösung:

```
/* include-Datei meine.h vom 16. 8.91                          */
#ifndef _MEINE_H

#define _MEINE_H

#define PI    3.141592654          /* Kreiszahl                 */
#define EU    2.718281828          /* Eulersche Zahl            */
int wieoft;                        /* eine globale Variable     */

#endif
```

Beim ersten vorkommenden `#include ¨meine.h¨` ist das Makro `_MEINE_H` noch nicht definiert; dadurch ist die `#ifndef`-Bedingung erfüllt und der Rest der Datei bis zum `#endif` wird verwertet; unter anderem wird dabei das Makro `_MEINE_H` definiert (ohne Wert).

Bei einem eventuell nachfolgenden `#include ¨meine.h¨` im selben Quelltext ist die `#ifndef`-Bedingung nicht mehr erfüllt, und die Datei meine.h wird bis zum `#endif` ignoriert.

17 Fehlersuche mit Standard-*C*

Im Sprachumfang von Standard-*C* ist eine portable, einfache und sehr praktische Hilfe zur Fehlersuche vorgesehen: ein Makro namens `assert()`.

`assert()` ist in assert.h definiert. Es wird nur expandiert, wenn `NDEBUG` nicht definiert ist.

Mit `assert()` kann man Bedingungen überprüfen, die zur Laufzeit *logisch wahr* sein sollen und ansonsten auf Programmfehler hinweisen (sogenannte Invarianten).

Falls die angegebene Bedingung entgegen aller Erwartung nicht erfüllt ist, wird das Programm abgebrochen.

Nach Entfernung der Programmfehler setzt man `#define NDEBUG` ein und hat damit alle Überprüfungen ausgeschaltet; die `assert()`-Aufrufe brauchen nicht entfernt zu werden.

Beispiel: Das Programm assdemof.c

```
/* assdemof.c 30. 7.95 kw
 */

#include <assert.h>
#include <stdio.h>

#define IMAX            10

int feld[IMAX];

/* zeige( index ) zeigt den Wert von feld[index]:
 */
void zeige( int index )
{
  assert( index<IMAX );  /* vergewissern ob index innerhalb */
  assert( index>=0   );  /* der Feldgrenzen liegt           */
  printf( "feld[%d] = %d\n", index, feld[index] );
}

main()
{
  int i;

  for( i=0; i<IMAX; i++ )
  {
    feld[i] = 2*i;
  }

  zeige( -2 );
  zeige( 5 );
  zeige( 10 );
}
```

enthält zwei Fehler. Sowohl mit `zeige( -2 )` als auch mit `zeige( 10 )` wird versucht, außerhalb der zulässigen Feldgrenzen auf `feld[]` zuzugreifen.

Stattdessen wollte der Programmierer eigentlich den 2., den 5. und den letzten (also den 9.) Wert des Feldes wissen.

Das Programm erzeugt in dieser Form nach dem Übersetzen, Linken und Starten die Ausgabe

```
assdemof: assdemof.c:16: zeige: Assertion ‘index>=0’ failed.
```

und bleibt stehen, weil in Zeile 16 die Bedingung `index>=0` verletzt wurde[32].

Angenommen, man kommt durch diesen Hinweis auf den Tippfehler `zeige( -2 )` und korrigiert die Zeile zu `zeige( 2 )`, dann erhält man –nach erneutem Kompilieren– die Ausgabe

```
feld[2] = 4
feld[5] = 10
assdem2f: assdem2f.c:15: zeige: Assertion ‘index<10’ failed.
```

und stellt vielleicht fest, daß das letzte Feldelement nicht mit `zeige( 10 )`, sondern vielmehr mit `zeige( 9 )` oder besser mit `zeige( IMAX-1 )` auszugeben ist.

Nach der Korrektur hat man das Programm

```
/* assdemok.c 30. 7.95 kw
 */

#include <assert.h>
#include <stdio.h>

#define IMAX            10

int feld[IMAX];

/* zeige( index ) zeigt den Wert von feld[index]:
 */
void zeige( int index )
{
  assert( index<IMAX );  /* vergewissern ob index innerhalb */
  assert( index>=0   );  /* der Feldgrenzen liegt           */
  printf( "feld[%d] = %d\n", index, feld[index] );
}

main()
{
  int i;

  for( i=0; i<IMAX; i++ )
  {
    feld[i] = 2*i;
  }
```

[32]Interessant ist daran, daß nicht nur das Programm mit einer Fehlermeldung abgebrochen wird, sondern auch die Stelle im Quelltext und der Wortlaut der verletzten Bedingung (!) ausgegeben wird. Dies ist allein durch den Praeprozessor möglich! Wer das trivial findet, kann zur Übung das Makro `assert()` selbst schreiben, ohne in `assert.h` zu spicken.

```
  zeige( 2 );
  zeige( 5 );
  zeige( IMAX-1 );
}
```

und erhält die Ausgabe:

```
feld[2] = 4
feld[5] = 10
feld[9] = 18
```

Damit kann das Programm als fehlerfrei angesehen werden und durch Einfügen der ersten Zeile:

```
#define NDEBUG
#include <assert.h>
#include <stdio.h>
...
```

werden alle `assert()`-Aufrufe ungültig und stören nicht weiter. Insbesondere erzeugen sie nicht mehr Programmcode, als wenn sie gar nicht vorhanden wären.

Bei später auftauchenden Fehlern kann man alle `assert()`-Aufrufe wieder aktivieren, indem man die Zeile `#define NDEBUG` entfernt (oder auskommentiert) und den Quelltext neu kompiliert.

18 Übungen

In den folgenden beiden Abschnitten sind mit jeweils gleichen Nummern eine Denksportaufgabe und eine oder mehrere Musterlösungen angegeben. Nur die Lösungen, welche diesen Rahmen wegen ihrer Größe sprengen würden, sind nicht abgedruckt. Bei Bedarf bitte auf Diskette anfordern.

18.1 Aufgaben

18.1.1 Unser erstes

Ein Programm ist zum Laufen zu bringen, welches den Text `Hallo!` auf den Bildschirm schreibt.

18.1.2 Zeilenvorschub

Für diese Aufgabe ist die Funktion `printf()` nötig; diese ist in stdio.h definiert.

Der Text `Hallo!` ist in einer Zeile auf den Bildschirm zu schreiben, danach sollen 2 Leerzeilen kommen und in der 4. Zeile der Text: `␣␣␣␣Ich␣bin␣Dein␣dummer␣Rechner!`.

Die ganze Ausgabe soll also so aussehen:

```
Hallo!


␣␣␣␣Ich␣bin␣Dein␣dummer␣Rechner!
```

Das Zeichen ␣ soll dabei ein Leerzeichen darstellen; die letzte Zeile sollte also mit 4 Leerzeichen beginnen.

18.1.3 Ganze Zahlen ausgeben (1)

Die Quadratzahlen (1, 4, 9, 16, usw.) bis 100 sind in jeweils einer Zeile auf den Bildschirm zu schreiben.

18.1.4 Ganze Zahlen ausgeben (2)

Die Zahlenfolge -1, 1, -4, 4, -9, 9, -16, 16 ist analog zur vorherigen Aufgabe auszugeben.

18.1.5 Tage im Monat

In dem folgenden Fragment ist die Funktion `tage()` zu ergänzen:

```
#include <stdio.h>

int tage( int monat )
{
  /* ??? */
}
```

```
main()
{
  printf( "Januar hat %d Tage\n",     tage(1)  );
  printf( "Februar hat %d Tage\n",    tage(2)  );
  printf( "Maerz hat %d Tage\n",      tage(3)  );
  printf( "April hat %d Tage\n",      tage(4)  );
  printf( "Mai hat %d Tage\n",        tage(5)  );
  printf( "Juni hat %d Tage\n",       tage(6)  );
  printf( "Juli hat %d Tage\n",       tage(7)  );
  printf( "August hat %d Tage\n",     tage(8)  );
  printf( "September hat %d Tage\n",  tage(9)  );
  printf( "Oktober hat %d Tage\n",    tage(10) );
  printf( "November hat %d Tage\n",   tage(11) );
  printf( "Dezember hat %d Tage\n",   tage(12) );
}
```

Schaltjahre brauchen nicht berücksichtigt zu werden.

`tage()` soll also beim Aufruf mit dem Argument 1 den Wert 31 liefern, beim Aufruf mit 2 den Wert 28 usw..

18.1.6 strcpy() selbst schreiben

Bei dieser und den folgenden Aufgaben werden Zeiger benötigt. Weiterhin muß man wissen, wie in *C* ein String aussieht (abschließende Null!).

Die Funktion

```
char *strcpy( char *wohin, const char *woher );
```

ist zu programmieren und auszutesten.

strcpy() (Seite 225) kopiert ab `woher` Zeichen nach `wohin`, bis das Stringende von `woher` erreicht ist, also bis ein Nullbyte kopiert wurde. Zurückgeliefert wird ein Zeiger auf die erzeugte Zeichenkette, also der Wert von `wohin`.

18.1.7 strcat() selbst schreiben

Die Funktion

```
char *strcat( char *wohin, const char *woher );
```

ist zu programmieren und auszutesten.

strcat() (Seite 224) kopiert ab `woher` Zeichen hinter das letzte Zeichen des Strings ab `wohin`, bis das Stringende von `woher` erreicht ist, also bis ein Nullbyte kopiert wurde. Zurückgeliefert wird ein Zeiger auf die erzeugte Zeichenkette, also der Wert von `wohin`.

18.1.8 Pascalstrings

Mit einem Datentyp

```
typedef char        pstring[81];
```

lassen sich *Pascal*-ähnliche Strings nachbilden.

Das nullte Element speichert dabei die aktuell verwendete Länge; in den Positionen 1 bis maximal 80 wird der eigentliche String gehalten.

Solche Strings dürfen im Gegensatz zu *C*-Strings auch Nullbytes speichern, da dieses Zeichen nicht als Stringende benötigt wird.

Dementsprechend können die Funktionen aus der *C*-Bibliothek nicht zur Manipulation der `pstring`s verwendet werden.

Aufgabe ist es, folgende Funktionen zu schreiben:

- `pstring *cstr_pstr( pstring ps, const char *cs );`

  kopiert einen *C*-String `cs` bis zur abschließenden 0 in einen Pascalstring `ps`.

  Nach dem Kopieren der eigentlichen Zeichen an die Stelle `ps[1]` und folgende muß natürlich noch die Anzahl der kopierten Zeichen in `ps[0]` eingetragen werden.

  Zurückgegeben wird ein Zeiger auf den erzeugten Pascalstring, also `ps`.

- `char *pstr_cstr( char *cs, const pstring ps );`

  kopiert den Inhalt eines Pascalstrings in eine normalen *C*-String und hängt dabei die Null am Stringende an.

  Rückgabe ist ein Zeiger auf den erzeugten *C*-String, also `cs`.

- `pstring *pstrcpy( pstring wohin, pstring woher );`

  kopiert den String `woher` nach `wohin`.

  Rückgabe ist ein Zeiger auf den neuen String, also `wohin`.

- `pstring *pstrcat( pstring woran, pstring woher );`

  hängt an den String `woran` den String `woher` an.

  Rückgabe ist ein Zeiger auf den neuen String, also `woran`.

- `int psprintf( pstring ps, const char *format, ... );`

  ist identisch mit den anderen `printf()`-Versionen, schreibt aber das Ergebnis in einen Pascalstring `ps` (*Achtung: Die Lösung ist nur einige Zeilen lang!*).

Die Funktionen brauchen keine Fehlerüberprüfung.

18.1.9 Stringausgabe

Für diese Aufgabe benötigt man die Funktion `putchar()` aus der Standardbibliothek und etwas Kenntnisse von Zeigern und Strings.

Das folgende Programm ist um den fehlenden Rest der Funktion `schreibwas()` zu ergänzen:

```
/* Hier ist putchar() vereinbart:
 */
#include <stdio.h>

/* Der String ab der Stelle s wird zeichenweise auf dem
 * Bildschirm ausgegeben, anschließend ein Zeilenvorschub.
 */
void schreibwas( char *s )
{

        /* ???   */
}

/* Testprogramm.
 */
main()
{
       schreibwas( "hallo!" );
       schreibwas( "nochmal hallo!" );
}
```

`schreibwas()` soll den übergebenen String zeichenweise auf den Bildschirm ausgeben und anschließend einen Zeilenvorschub erzeugen, also genauso arbeiten wie die Funktion `puts()` aus der Standardbibliothek. `puts()` ist für die Lösung dieser Aufgabe aber nicht zu benutzen! Vielmehr soll der String zeichenweise ausgegeben werden, bis die abschließende Null erreicht ist.

18.1.10 Struktur für 5 Personen

Nötige Kenntnisse: Strings, Felder, Strukturen, `printf()` und `scanf()` aus der Standardbibliothek.

Zu schreiben ist ein kleines Programm, welches eine Struktur als Datentyp vereinbart zum Speichern einer Person. Eine Person soll aus Vorname, Nachname und Alter bestehen.

Von diesem Datentyp ist ein Feld für 5 Personen zu definieren.

Weiterhin soll eine Funktion enthalten sein, welche die Daten einer Person von der Tastatur einliest, und eine Funktion, welche die Daten einer Person ausgibt.

Das Hauptprogramm liest in einer Schleife mit der erstgenannten Funktion alle 5 Personen ein. Danach erst werden mit der zweiten Funktion alle 5 Personen wieder ausgegeben.

18.1.11 Struktur für 10 Personen

Wie die vorhergehende Aufgabe **18.1.10 Struktur für 5 Personen** auf Seite 136, aber für 10 Personen!

18.1.12 lsearch()

Für diese Aufgabe muß man etwas mit Zeigern umgehen können, außerdem braucht man *casts* für Typumwandlungen, Zeiger auf Funktionen und Felder.

Das Beispiel zum Aufruf der zu schreibenden Funktion verwendet Strukturen; für die Aufgabe selbst ist deren Kenntnis aber unwichtig.

Eine Funktion mit der Deklaration

```
void *lsearch( const void *schluessel,
               const void *feldanfang,
               size_t elemente,
               size_t elementgroesse,
               int(* vergleichsfkt)( const void *e1,
                                     const void *e2
                                   )
             );
```

ist zu schreiben. Die Funktion soll in einem eventuell unsortierten Feld ab `feldanfang` mit linearer Suche ein Element finden, für das die Funktion `(*vergleichsfkt)()` beim Vergleich mit dem Element `(*schluessel)` den Wert `0` liefert, also Gleichheit.

Wird ein solches Element gefunden, dann wird ein Zeiger darauf zurückgegeben. Wird kein Element gefunden, dann soll der `NULL`-Zeiger zurückgegeben werden.

Der Aufruf ist identisch mit dem von `bsearch()` aus der Standardbibliothek.

Das Feld ab `feldanfang` muß `(elemente)` Werte beinhalten; jedes Element hat die Größe `(elementgroesse)`.

`vergleichsfkt` zeigt auf eine Funktion, die man mit 2 Zeigern als Parameter aufrufen kann und die ein Resultat vom Typ `int` liefert. Wenn die Parameter auf Daten von dem Typ zeigen, den auch jedes Element in dem Feld ab `feldanfang` hat, dann muß diese Funktion den Wert `0` liefern, wenn die beiden Daten als gleich zu betrachten sind.

Für den Parameter `vergleichsfkt` kann man also gegebenfalls die gleiche Funktion verwenden wie für `bsearch()` und `qsort()`.

Der Rückgabewert von `lsearch()` ist ein Zeiger auf das gefundene Element, oder `NULL`, wenn in dem Feld kein Element steht, für das `(*vergleichsfkt)` beim Vergleich mit dem Schlüssel `(*schluessel)` den Wert `0` liefert.

`lsearch()` soll ebenso wie `qsort()` oder `bsearch()` für beliebige Datentypen funktionieren, da die Behandlung der Daten (Feld und Schlüssel) nur in der übergebenen Vergleichsfunktion erfolgt, die wiederum vom Aufrufer zur Verfügung zu stellen ist.

`lsearch()` soll von der Funktion und vom Aufruf her identisch sein mit `bsearch()`, außer daß das Feld für `bsearch()` sortiert sein muß. Für `lsearch()` ist dies nicht nötig.

Beispiel: Es wird in einem Programm eine Struktur gebildet, in der ein Autoname und der zugehörige Intelligenzquotient des Fahrers Platz finden. Dieser Datentyp soll `Auto_t` heißen:

```
typedef struct
{
    char    name[30];   /* Autoname          */
    int     iq;         /* IQ des Fahrers    */
}
Auto_t;
```

In dem Beispielprogramm wird von diesem Typ ein Feld vereinbart und mit einigen Werten initialisiert.

Der Benutzer wird beim Programmlauf aufgefordert, einen Autonamen einzugeben. Dann sucht das Programm mit der noch unbekannten Funktion `lsearch()` nach dem Feldelement, in dem der eingegebene Autonamen enthalten ist und gibt den zugehörigen IQ aus.[33]

```
/* aulsear.c 30. 7.95 kw
 * Beispielprogramm fuer lsearch().
 */

#include <string.h>

/* lsearch() deklarieren:
 */
void *lsearch( const void *schluessel,
               const void *feldanfang,
               size_t elemente,
               size_t elementgroesse,
               int(* vergleichsfkt)( const void *e1,
                                     const void *e2
                                     )
               );

/* Den Datentyp Auto_t vereinbaren:
 */
typedef struct
{
  char    name[30];   /* Autoname         */
  int     iq;         /* IQ des Fahrers   */
}
Auto_t;

/* Davon ein Feld vereinbaren und initialisieren:
 */
#define N       4
Auto_t  Auto[N] =
{   { "525",   40 },
    { "190",   20 },
    { "Manta", 0  },
    { "Polo",  80 }
};

/* Die fuer lsearch() noetige Vergleichsfunktion erhaelt
 * zwei Zeiger auf Daten, in unserem Fall vom Typ Auto_t.
 * Sind die darin enthaltenen Autonamen gleich, dann
 * soll der Wert 0 geliefert werden.
 * Die Unterscheidung zwischen "groesser" und "kleiner"
 * ist fuer lsearch() nicht noetig, wird aber so gleich
 * mitgeliefert:
 */
int verglname( const void *Auto1, const void *Auto2 )
```

[33]Eventuelle Ähnlichkeiten mit dem realen Straßenverkehr sind rein zufällig.

```
{
  return strcmp( ((Auto_t*)Auto1)->name,
                 ((Auto_t*)Auto2)->name  );
}


/* Das Hauptprogramm liest von der Standardeingabe einen
 * Autonamen und versucht, den IQ dazu zu finden:
 */

int main()
{
  Auto_t
    suchauto,               /* danach suchen            */
    *gefunden;              /* Zeiger auf gefundenes    */
                                /* Auto                     */
  /* Den Autonamen lesen:
   */
  puts( "Bitte einen Autonamen:" );
  gets( suchauto.name );

  /* suchauto dient fuer lsearch() als Schluessel.
   * Da in der Vergleichsfunktion verglname() nur
   * der Name verwendet wird, ist der iq in suchauto
   * egal.
   */

  /* Jetzt mit lsearch() einen Zeiger auf das Element
   * holen, das den gleichen Namen enthaelt wie
   * suchauto:
   */
  gefunden = lsearch( &suchauto,
                      Auto,
                      (size_t)N,
                      sizeof(Auto_t),
                      verglname       );

  /* Wenn gefunden ungleich NULL ist, dann wurde ein
   * Element gefunden:
   */
  if( gefunden )
    printf( "Der IQ eines %s-Fahrers ist etwa %d\n",
            gefunden->name,
            gefunden->iq
            );
  else
    printf( "%s-Fahrer kenne ich nicht.\n",
            suchauto.name
            );

  return 0;
}
```

Gesucht ist also die Funktion `lsearch()`.

18.1.13 Zahlen aus Datei lesen

Voraussetzungen: Öffnen von Dateien und Lesen daraus.

Aus der Textdatei werte.dat, die so aussehen kann:

```
4711
4712
42
0
0
4713
```

sollen Zahlenwerte vom Typ `int` gelesen werden, bis das Dateiende erreicht wird.

Das zu schreibende Programm soll die Zahlen einlesen, die Anzahl der gelesenen Werte ausgeben sowie den Durchschnitt der Zahlen.

18.1.14 Operatoren auf die Schnelle

Welchen Wert hat in *C* der Ausdruck `-1<0<1` ?

18.1.15 Mit Zeigern jonglieren, aber richtig! (I)

Hierfür muß man mit Zeigern gut umgehen können.

Die Aufgabe lautet: Was macht das folgende Programm[34]? Kann es überhaupt richtig funktionieren?

```
/* aujong1.c 30. 7.95 kw
 * Was macht dieses Programm?
 */

char *c[]  =
{
  "he dast ga",
  "lllt dumm",
  "C i",
  "dar nich"
};

char **cp[] =
{
  c+3,
  c+2,
  c+1,
  c
};


char ***cpp  =   cp;
```

[34]Der hier gezeigte Quelltext ist naturgemäß kein Musterbeispiel an Lesbarkeit und Übersichtlichkeit. Zur Nachahmung ist dieser Stil sicher nicht empfohlen!

```
main( int Anzarg, char **Arg )
{
  printf( "%s", **++cpp );
  printf( "%s", *--*++cpp+5 );
  printf( "%s", cpp[-2][0]+2 );
  printf( "%s\n", *(cpp[1]+1)+3 );
  getchar();
}
```

Das Programm soll nicht abgetippt und ausprobiert werden, bis die Lösung gefunden ist! Vielmehr soll die Lösung nur durch Überlegen gefunden werden. Das Aufzeichnen der Variablen auf einem Blatt Papier erleichtert die Lösung enorm.

Noch mehr Aufgaben dieser Art findet man in [C-Puzzle].

18.1.16 Mit Zeigern jonglieren, aber richtig! (II)

Für diese Aufgabe muß man etwas mit Zeigern umgehen können, außerdem braucht man *casts* für Typumwandlungen, Zeiger auf Funktionen und Felder. Schließlich muß man auch die **Rekursion** (Seite 95) verstanden haben.

Aufgabe ist, die Funktion **qsort()** (Seite 213) aus der Standardbibliothek selbst zu schreiben.

Zum Testen kann das Beispielprogramm zu **qsort()** (Seite 214) zweckentfemdet werden.

Der *quick sort*-Algorithmus sortiert ein Feld folgendermaßen:

- Das Feld wird so umsortiert, daß ein beliebig herausgegriffenes Feldelement oberhalb aller kleineren, aber unterhalb aller größeren Elemente zu liegen kommt.
- Der Algorithmus wird rekursiv auf alle Elemente links vom gewählten Element angewandt, dann
- auf alle Element rechts davon.

Die Rekursion bricht ab, wenn ein Teilfeld weniger als zwei Elemente hat.

Zur Auswahl des Elementes: Am günstigsten wäre es, das Element so zu wählen, daß die entstehenden Teilfelder etwa gleich groß sind. Da aber dieses Element nicht bekannt ist, wählt man einfach das Element in der Feldmitte.

18.1.17 Kleiner Praeprozessor

Zu schreiben ist ein Programm **pp**, welches einige Funktionen des Präprozessors von *C* ausführt.

Folgende Funktionen sollten mindestens enthalten sein:

- die `#include` ¨. . . ¨-Anweisung.
- die `#include <...>`-Anweisung. Dabei sollte der Suchpfad aus der Environmentvariablen `INCLUDE` gelesen werden. Der Inhalt dieser Variablen läßt sich mit der Funktion **getenv()** (Seite 199) bestimmen.

- die `#define`-Anweisung ohne Parameter.
- die `#undef`-Anweisung.
- die Anweisungen `#if`, `#if defined`, `#ifdef`, `#ifndef`, `#elif`, `#else` und `#endif`.

  Achtung! Die `#if`-Konstruktionen sollten geschachtelt werden können.

Als Bonusaufgabe darf das Programm auch gerne die `#define`-Anweisung mit Parametern beherrschen!

Die anderen Direktiven sowie die Spezialitäten `#` und `##` brauchen nicht vorgesehen zu werden.

Das Programm soll mit zwei Parametern aufgerufen werden.

- Der erste Parameter gibt an, welche Datei gelesen werden soll.
- Der zweite Parameter ist der Name der Ausgabedatei.

Der zu schreibende Praeprozessor liest also eine Datei und erzeugt in einer weiteren Datei eine Ausgabe. Nach den üblichen Regeln des *C*-Praeprozessors sollen dabei mit `#include` weitere Dateien in die Ausgabe eingefügt werden, mit `#define` definierte Namen durch ihren Wert ersetzt werden, und so fort.

Die mit `#include` eingefügten Dateien durchlaufen wiederum den Praeprozessor.

Da das fertige Programm unabhängig vom *C*-Compiler ist, kann man es genausogut zum Entwickeln von *FORTRAN*- oder *Pascal*-Programmen oder zur Arbeit mit TEX verwenden.

Auf Fehleingaben (zu wenige, zu viele Parameter, falsche Namen etc.) soll das Programm sinnvoll reagieren (Erklärung ausgeben, abbrechen...).

18.2 Lösungen

Die angegebenen Lösungen sind natürlich nicht die einzig möglichen; bei den meisten Aufgaben kann man ebenso Dutzende weitere sinnvolle Lösungen finden. Von diesem Kapitel kann man sich also höchstens Anregungen holen oder eigene Lösungen mit den hier angegebenen vergleichen.

18.2.1 Unser erstes

```
/* loerst1.c 30. 7.95 kw
 * Musterloesung zur ersten Aufgabe
 */

/* die verwendete Funktion deklarieren
 * (nicht unbedingt noetig, aber zweckmaessig):
 */
int puts( char *string );

/* Hier beginnt das Hauptprogramm:                          */
main()
{
  puts( "Hallo!" );
}
```

18.2.2 Zeilenvorschub

```
/* lozeil.c 30. 7.95 kw
 */

/* puts() deklarieren:
 */
int puts( const char *string );

main()
{
  puts( "Hallo!\n\n\n    Ich bin Dein dummer Rechner!" );
}
```

18.2.3 Ganze Zahlen ausgeben (1)

```
/* loganz1.c 30. 7.95 kw
 */

/* printf() vereinbaren:
 */
#include <stdio.h>

/* Hier wird ein Schleifenzaehler i vereinbart.
 * Dieser soll von 1 bis 10 laufen; sein Quadrat wird mit
 * printf() ausgegeben.
 */
main()
```

```
{
  int i;      /* Schleifenzaehler                      */

  for( i=1; i<=10; i=i+1 )   /* Schleife von 1 bis 10 */
    printf( "Das Quadrat von %d ist %d\n",
            i,
            i*i
            );
}
```

18.2.4 Ganze Zahlen ausgeben (2)

```
/* loganz2.c 30. 7.95 kw
 */

/* printf() vereinbaren:
 */
#include <stdio.h>

/* Hier wird ein Schleifenzaehler i vereinbart.
 * Dieser soll von 1 bis 4 laufen; sein Quadrat wird mit
 * printf() negativ und positiv ausgegeben.
 */
main()
{
  int i;      /* Schleifenzaehler                      */

  for( i=1; i<=4; i=i+1 )   /* Schleife von 1 bis 4   */
    printf( "%d\n%d\n",
            -i*i,
            i*i
            );
}
```

18.2.5 Tage im Monat

Die direkteste Lösung wäre wahrscheinlich:

```
/* lotim1.c 30. 7.95 kw
 */

int tage( int monat )
{
  if( monat==1  ) return 31;
  if( monat==2  ) return 28;
  if( monat==3  ) return 31;
  if( monat==4  ) return 30;
  if( monat==5  ) return 31;
  if( monat==6  ) return 30;
  if( monat==7  ) return 31;
  if( monat==8  ) return 31;
  if( monat==9  ) return 30;
```

```
  if( monat==10 ) return 31;
  if( monat==11 ) return 30;
  if( monat==12 ) return 31;
  return 0;
}
```

Etwas kompakter könnte man schreiben:

```
/* lotim2.c 30. 7.95 kw
 */

int tage( int monat )
{
  if( monat==1 || monat==3  ||
      monat==5 || monat==7  ||
      monat==8 || monat==10 ||
      monat==12
      )
    return 31;
  if( monat== 2 )
    return 28;
  if( monat==4 || monat==6  ||
      monat==9 || monat==11
      )
    return 30;
  return 0;
}
```

Noch deutlich kürzer wird die Funktion `tage()` so:

```
/* lotim3.c 30. 7.95 kw
 */

int tage( int monat )
{
  int tagfeld[12] =
  {
    31, 28, 31, 30, 31, 30,
    31, 31, 30, 31, 30, 31
  };

  if( monat>=1 && monat<=12 )
    return tagfeld[monat-1];
  else
    return 0;
}
```

oder so:

```
/* lotim4.c 30 .7.95 kw
 */

int tage( int monat )
```

```
{
  int tagfeld[12] =
  {
    31, 28, 31, 30, 31, 30,
    31, 31, 30, 31, 30, 31
  };

  return ( monat>=1 && monat<=12 ? tagfeld[monat-1] : 0 );
}
```

18.2.6 strcpy() selbst schreiben

```
/* lostrcp1.c 30. 7.95 kw
 */

char *strcpy( char *wohin, const char *woher )
{
  char   *wohinkopie;     /* Kopie von wohin zum
                           * Zurueckgeben
                           */
  char    zeichen;        /* Zu kopierendes Zeichen
                           */
  /* wohin merken fuer Rueckgabewert:
   */
  wohinkopie = wohin;
  /* Kopieren, bis 0 auftritt:
   */
  do
  {
    zeichen = *woher;   /* Zeichen holen ...            */
    *wohin  = zeichen;  /* ... und schreiben.           */
    wohin = wohin + 1;  /* Beide Zeiger um eins ...     */
    woher = woher + 1;  /* ... erhoehen.                */
  }
  while (zeichen!=0);   /* Das ganze, bis die 0 kopiert
                         * wurde.
                         */
  /* Fertig.
   */
  return wohinkopie;
}

/* Testprogramm fuer strcpy()                            */
main()
{
  char    platz[50];  /* genug Platz zum Ausprobieren     */
  char   *zeiger;

  zeiger = strcpy( platz, "Teststring" );
  puts( zeiger );
}
```

Eine kompaktere, aber nicht so leicht verständliche Lösung wäre:

```
/* lostrcp2.c 30. 7.95 kw
 */

char *strcpy( char *wohin, const char *woher )
{
  char   *wohinkopie;     /* Kopie von wohin zum
                           * Zurueckgeben
                           */

  /* wohin merken fuer Rueckgabewert:
   */
  wohinkopie = wohin;

  /* Kopieren, bis 0 auftritt:
   */
  while( (*wohin++=*woher++) )
    ;       /* In der while-Bedingung wird schon alles
             * erledigt, also leere Anweisung.
             */
  /* Fertig.
   */
  return wohinkopie;
}

/* Testprogramm fuer strcpy()
 */
main()
{
  char    platz[50];  /* genug Platz zum Ausprobieren
                       */
  strcpy( platz, "Teststring" );
  puts( platz );
}
```

18.2.7 strcat() selbst schreiben

Die ausführliche Lösung wieder zuerst (der Quelltext von `strcpy()` ist aus Platzgründen weggelassen worden):

```
/* lostrca1.c 30. 7.95 kw
 */

char *strcpy( char *wohin, const char *woher )
     /* ... siehe Loesung der letzten Aufgabe ...
      */

     char *strcat( char *wohin, const char *woher )
{
  char   *wohinkopie;     /* Kopie von wohin zum
                           * Zurueckgeben
```

```
                         */
  char    zeichen;        /* Zu kopierendes Zeichen
                         */

  /* wohin merken fuer Rueckgabewert:
   */
  wohinkopie = wohin;

  /* wohin bis zur abschliessenden 0 erhoehen:
   */
  while( *wohin!=0 ) wohin = wohin + 1;

  /* Kopieren, bis 0 auftritt:
   */
  do
  {
    zeichen = *woher;   /* Zeichen holen ...            */
    *wohin  = zeichen;  /* ... und schreiben.           */
    wohin = wohin + 1;  /* Beide Zeiger um eins ...     */
    woher = woher + 1;  /* ... erhoehen.                */
  }
  while (zeichen!=0);   /* Das ganze, bis die 0 kopiert
                         * wurde.
                         */

  /* Fertig.
   */
  return wohinkopie;
}

/* Testprogramm fuer strcpy() und strcat()
 */
main()
{
  char    platz[50];  /* genug Platz zum Ausprobieren     */
  char   *zeiger;

  strcpy( platz, "Teststring" );
  strcat( platz, " mit Anhang" );
  puts( platz );
}
```

Und hier eine kompaktere Lösung:

```
/* lostrca2.c 30. 7.95 kw
 */

char *strcpy( char *wohin, const char *woher )
     ...;

     char *strcat( char *wohin, const char *woher )
{
  char   *wohinkopie;     /* Kopie von wohin zum
```

```
                          * Zurueckgeben
                          */

  /* wohin merken fuer Rueckgabewert:
   */
  wohinkopie = wohin;

  /* wohin auf die abschliessende 0 erhoehen:
   */
  while( *wohin ) wohin++;

  /* Kopieren, bis 0 auftritt:
   */
  while( (*wohin++=*woher++) )
    ;       /* In der while-Bedingung wird schon alles
             * erledigt, also leere Anweisung
             */
  /* Fertig.
   */
  return wohinkopie;
}

/* Testprogramm fuer strcpy() und strcat()
 */
main()
{
  char    platz[50];  /* genug Platz zum Ausprobieren     */
  char   *zeiger;

  strcpy( platz, "Teststring" );
  strcat( platz, " mit Anhang" );
  puts( platz );
}
```

18.2.8 Pascalstrings

```
/* lopasstr.c 30. 7.95 kw
 */

typedef char        pstring[81];


pstring *cstr_pstr( pstring ps, const char *cs );
char *pstr_cstr( char *cs, const pstring ps );
pstring *pstrcpy( pstring wohin, pstring woher );
pstring *pstrcat( pstring woran, pstring woher );
int psprintf( pstring ps, const char *format, ... );


pstring *cstr_pstr( pstring ps, const char *cs )
{
  char   *p = &ps[1];     /* Zeiger in Pascalstring    */
```

```
  ps[0] = 0;              /* Laenge zu 0 setzen        */
  while( *cs )            /* bis C-Stringende ...      */
  {
    *p++ = *cs++;         /* ... Zeichen kopieren      */
    ps[0]++;              /* und Laenge korrigieren.   */
  }
  return ps;              /* Pascalstring zurueckgeben */
}


char *pstr_cstr( char *cs, const pstring ps )
{
  char
    *p = &ps[1],          /* Zeiger in Pascal- und     */
    *c = cs;              /* C-string                  */
  int
    i = ps[0];            /* Schleifenzaehler          */

  while( i-- )
    *c++ = *p++;              /* i Zeichen kopieren    */
  *c = 0;                     /* Stringende            */
  return cs;                  /* C-String zurueckgeben */
}


pstring *pstrcpy( pstring wohin, pstring woher )
{
  int     i = woher[0] + 1;   /* Schleifenzaehler      */
  char   *p = wohin;

  while( i-- ) *p++ = *woher++;  /* i Zeichen kopieren */
  return wohin;
}


pstring *pstrcat( pstring woran, pstring woher )
{
  int     i = woher[0];          /* Schleifenzaehler   */
  char   *p = woran+woran[0];    /* Ende von woran     */

  *woran += i;                   /* Laenge korrigieren */
  while( i-- ) *++p = *++woher;  /* i Zeichen kopieren */
  return woran;
}


/* Fuer psprintf() wird die Funktion vsprintf() aus der
 * Standardbibliothek verwendet. Diese macht die ganze
 * Arbeit, ausser dass man von ihr den Pascalstring ab ps[1]
 * eintragen lassen muss.
 * vsprintf() liefert die Anzahl der Zeichen als
 * Rueckgabewert; diese Zahl kann man in ps[0] eintragen
 * und ist damit schon fertig.
 */
```

```
#include <stdarg.h>

int psprintf( pstring ps, const char *format, ... )
{
  va_list     args_p;
  va_start( args_p, format );

  ps[0] = vsprintf( &ps[1], format, args_p );
  va_end( args_p );
  return ps[0];
}


/* Testprogramm fuer die Pascalstringfunktionen.
 */
int main()
{
  pstring     ap, bp;
  char        ac[80];

  cstr_pstr( ap, "Das ist ein Teststring" );

  pstr_cstr( ac, ap );
  printf( "ac ist [%s]\n", ac );

  cstr_pstr( bp, " und mehr" );
  pstrcat( ap, bp );

  pstr_cstr( ac, ap );
  printf( "ac ist [%s]\n", ac );

  psprintf( ap,
            "%d, %d, %d; Mandela ist %s",
            1,
            2,
            3,
            "frei"
            );

  pstr_cstr( ac, ap );
  printf( "ac ist [%s]\n", ac );
}
```

18.2.9 Stringausgabe

```
/* lostr.aus 30. 7.95 kw
 *
 * Musterloesung zur Stringausgabe.
 */

/* Hier ist putchar() vereinbart:
```

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


/* Das ist die zu schreibende Funktion.
 * Der String ab der Stelle s wird zeichenweise auf dem
 * Bildschirm ausgegeben, anschliessend ein Zeilenvorschub.
 */
void schreibwas( char *s )
{
  while( *s ) putchar( *s++ );  /* String ausgeben.   */
  putchar( '\n' );              /* Zeilenvorschub.    */
}


/* Testprogramm.                                      */
main()
{
  schreibwas( "hallo!" );
  schreibwas( "nochmal hallo!" );
}
```

18.2.10 Struktur für 5 Personen

```
/* lostru5p.c 30. 7.95 kw
 * Musterloesung zum Lesen und Ausgeben von Personen.
 */

/* Vereinbarungen zur Ein- und Ausgabe:
 */
#include <stdio.h>

/* Soviele Personen sollen es sein:
 */
#define         N_PERSONEN    5


/* Datenstruktur fuer eine Person:                          */
typedef struct
{
  char      vname[40];
  char      nname[40];
  int       alter;
}
person_t;


/* Ab hier kann man mit dem Datentypen person_t Variablen
 * vereinbaren.
 */

/* Funktion zum Lesen einer Person von der Tastatur:
```

```
 * Da lese_person() eine Struktur vom Typ person_t aendern
 * soll, muss man einen Zeiger auf die Struktur uebergeben.
 * Deshalb hat der Parameter den Typ person_t
 * Dementsprechend kann man nicht mit p.vname etc. auf die
 * Elemente der Struktur zugreifen, sondern nur mit
 * (*p).vname etc.; dafuer gibt es die abkuerzende Schreib-
 * weise p->vname.
 * An scanf() muessen die Adressen uebergeben werden, an
 * welche die eingelesenen Wert geschrieben werden sollen.
 * Deshalb das &-Zeichen vor p->alter.
 * Bei den Elementen vname und nname ist der Adressoperator
 * & nicht noetig, da diese Elemente Felder sind
 * (char ...[40]) und Felder ohnehin als Adressen ueber-
 * geben werden.
 */

void lese_person( person_t *p )
{
  printf( "Bitte Vorname, Nachname und Alter: " );
  scanf( "%s%s%d", p->vname, p->nname, &(p->alter) );
}


/* Funktion zum Schreiben einer Person:
 */
void schreibe_person( person_t p )
{
  printf( "%s %s ist %d Jahre alt.\n",
          p.vname,
          p.nname,
          p.alter
          );
}


main()
{
  int         i;
  person_t    personen[N_PERSONEN];

  /* Die Personen einlesen:
   */
  printf( "Bitte %d Personen eingeben!\n",
          N_PERSONEN
          );
  for( i=0; i<N_PERSONEN; i++ )
    lese_person( &personen[i] );

  /* Alle Personen wieder ausgeben:
   */
  printf( "Sie hatten eingegeben:\n" );
  for( i=0; i<N_PERSONEN; i++ )
    schreibe_person( personen[i] );
}
```

18.2.11 Struktur für 10 Personen

Wenn man die vorhergehende Aufgabe sauber gelöst hat, also für die Anzahl der Personen überall im Programm nur eine einzige Praeprozessorkonstante verwendet hat, dann braucht man jetzt auch nur diese einzige `#define`-Anweisung ändern:

```
#define         N_PERSONEN    5
```

ist zu ändern in:

```
#define         N_PERSONEN    10
```

und fertig.

18.2.12 lsearch()

```
/* lolsear.c 30. 7.95 kw
 * Musterloesung fuer lsearch()
 */

void *lsearch( const void *schluessel,
               const void *feldanfang,
               size_t elemente,
               size_t elementgroesse,
               int(* vergleichsfkt)( const void *e1,
                                     const void *e2
                                     )
               )
{
  char       *feld_p = (char *)feldanfang;

  /* Parameter pruefen:
   */
  if( !( schluessel
         && feldanfang
         && elemente
         && elementgroesse
         && vergleichsfkt   )      ) return NULL;

  /* Schleife ueber alle Elemente:
   */
  while( elemente-- )
  {
    /* trifft das aktuelle Element zu?
     */
    if( !(*vergleichsfkt)( schluessel, feld_p ) )
      return feld_p;
    /* Wenn nicht, dann naechstes Element:
     */
    feld_p += elementgroesse;
  }

  /* Ende des Feldes und trotzdem nichts gefunden?
   */
  return NULL;
}
```

18.2.13 Zahlen aus Datei lesen

```
/* lozahdat.c 30. 7.95 kw
 * Musterloesung zum Lesen von Zahlen aus werte.dat
 */

#include <stdio.h>

#define DATEINAME       "werte.dat"


int main()
{
  FILE      *f;            /* Datei, aus der gelesen
                            * werden soll
                            */
  long int
    summe=0,               /* Zum Aufsummieren           */
    wert;                  /* gelesener Wert             */
  int         anzahl=0;    /* Anzahl gelesener Werte     */

  if( (f=fopen( DATEINAME, "r" )) )      /* Datei oeffnen */
  {
    while( 1==fscanf( f, "%ld", &wert ) ) /* Lesen  */
    {
      summe += wert;       /* Summieren */
      anzahl++;            /* Anzahl merken */
    }

    fclose( f );           /* Datei schliessen */
    /* Ergebnis ausgeben:                      */
    printf( "%d Werte gelesen; Mittel ist %f\n",
            anzahl, (double)summe/anzahl
            );
  }
  else
    puts( "Ich kann " DATEINAME "nicht oeffnen!" );

  return 0;
}
```

18.2.14 Operatoren auf die Schnelle

`-1<0<1` ist ein ganzzahliger Ausdruck mit dem Wert 0.

Begründung: `-1<0<1` besteht aus zwei Vergleichen (mit `<` für *kleiner*). Da Ausdrücke mit dem Operator `<` von links her zusammengefaßt werden (siehe Tabelle auf Seite 83), ist der gegebene Ausdruck identisch mit `(-1<0)<1`. Der innere Teilausdruck davon (`(-1<0)`) ist ein logischer Vergleich, der mit *wahr* bewertet wird, da -1 ja kleiner ist als 0. Ein erfüllter logischer Ausdruck liefert den Wert 1 (siehe Seite 39); wodurch der Gesamtausdruck als `1<1` geschrieben werden kann.

Dies ist wiederum ein Vergleich, der nicht erfüllt ist (1 ist nicht kleiner als 1) und deshalb den Wert 0 für *logisch falsch* liefert. Dies ist dann das Ergebnis des gesamten Ausdrucks.

18.2.15 Mit Zeigern jonglieren, aber richtig! (I)

Das angegebene Programm funktioniert, und zwar gibt es folgenden Text aus:

```
C ist gar nicht dumm
```

Danach wartet das Programm darauf, daß der Benutzer auf die Eingabetaste drückt.

Zur Erklärung:

Die Definition

```
char       *c[]  = { "he dast ga", "lllt dumm", "C i", "dar nich"
                   };
```

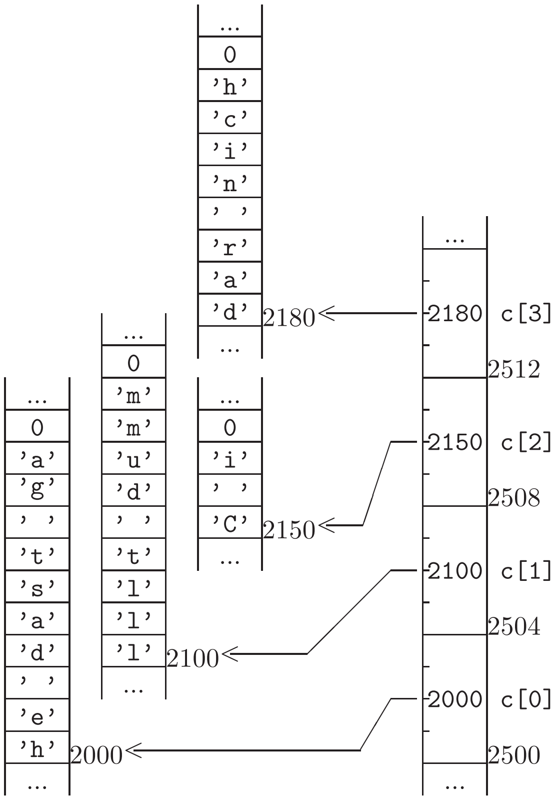
legt im Speicher 4 Stringkonstanten (Felder vom Typ `char`) an und vereinbart ein Feld `c` mit 4 Elementen vom Typ `char*`. Die Stringkonstanten liegen irgendwo im Speicher und sind alle mit je einem Nullbyte abgeschlossen. Die 4 Zeiger auf `char` (`c[0]` bis `c[3]`) zeigen jeweils auf das erste Zeichen einer Stringkonstante.

Die Stringkonstanten sollen in unserem Beispiel an den Adressen 2000, 2100, 2150 und 2180 im Arbeitsspeicher liegen.

Das Feld`c` möge bei 2500 beginnen.

Die Anzahl der Feldelemente in `c` ergibt sich aus der Anzahl der Initialisierungen.

Die Situation im Speicher (zur Laufzeit) sieht also etwa so aus:



`c[0]` hat in unserem Beispiel den Wert 2000 und liegt selbst an der Stelle 2500 im Speicher, `c[1]` den Wert 2100 und liegt bei 2504, `c[2]` ist 2150 und `c[3]` enthält 2180.

Dabei wird angenommen, daß eine `char`-Variable ein Byte belegt und jeder Zeiger 4 Byte. Neben den Speicherzellen stehen in der Zeichnung gegebenenfalls die zugehörigen Variablennamen und gedachte Adressen der Elemente im Speicher.

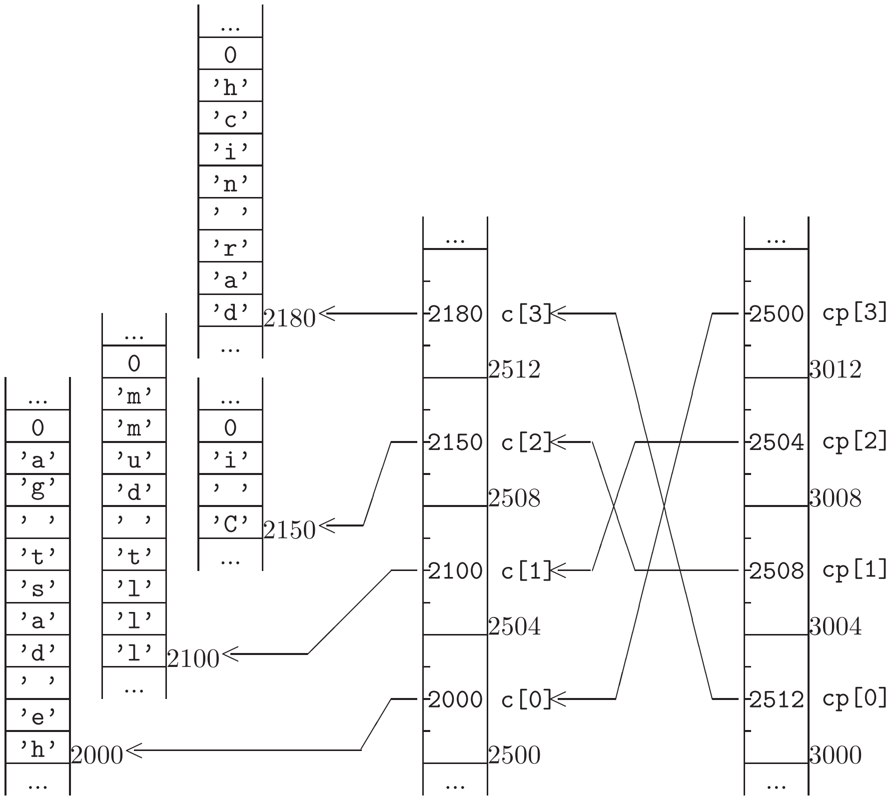
```
char      **cp[] = {   c+3, c+2, c+1, c
                   };
```

vereinbart ein Feld `cp` mit 4 Zeigern, die auf die eben erst angelegten Zeiger `c` zeigen; und zwar zeigt `cp[0]` auf `c[3]` (3 Elemente hinter die Anfangsadresse des Feldes `c`), `cp[1]` zeigt auf `c[2]` und so fort.

`cp[0]` hat in unserem Beispiel also den Wert 2512, `cp[1]` den Wert 2508, `cp[2]` ist 2504 und `cp[3]` enthält 2500.

Das Feld `cp` soll an der Speicherstelle 3000 beginnen.

Dadurch ergibt sich folgende Situation:



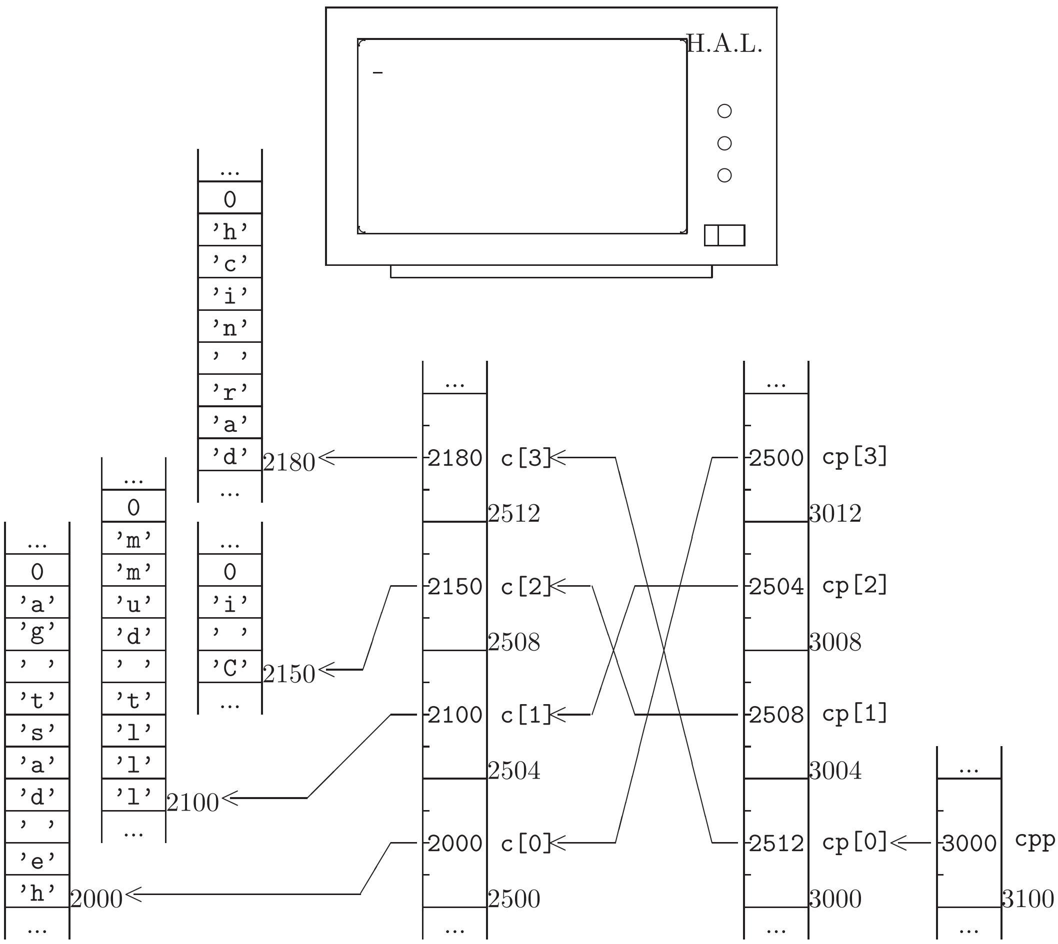
Die folgende Definition:

```
char     ***cpp   =   cp;
```

vereinbart einen Zeiger `cpp`, der auf ein Element vom Typ `char**` zeigt, nämlich auf `cp[0]`.

Da `cp[0]` in unserem Beispiel an der Stelle 3000 liegt, wird `cpp` mit der Adresse 3000 initialisiert.

Im Speicher könnte es also nun so aussehen:



Der Bildschirm soll (außer einem kleinen Cursor: _) noch leer sein.

Die Anweisung:

```
printf( "%s", **++cpp );
```

erhöhtcpp um 1, also zeigtcpp nun auf cp[1][35].

Mit „Erhöhung um 1“ ist nicht die Addition (3000+1) gemeint, sondern vielmehr die Operation (3000+1*sizeof(char**)), siehe dazu **Zeigerarithmetik** (Seite 46). cpp erhält also den neuen Wert 3004, weil wir für unser Beispiel 4 Byte pro Zeigervariable spendierten.

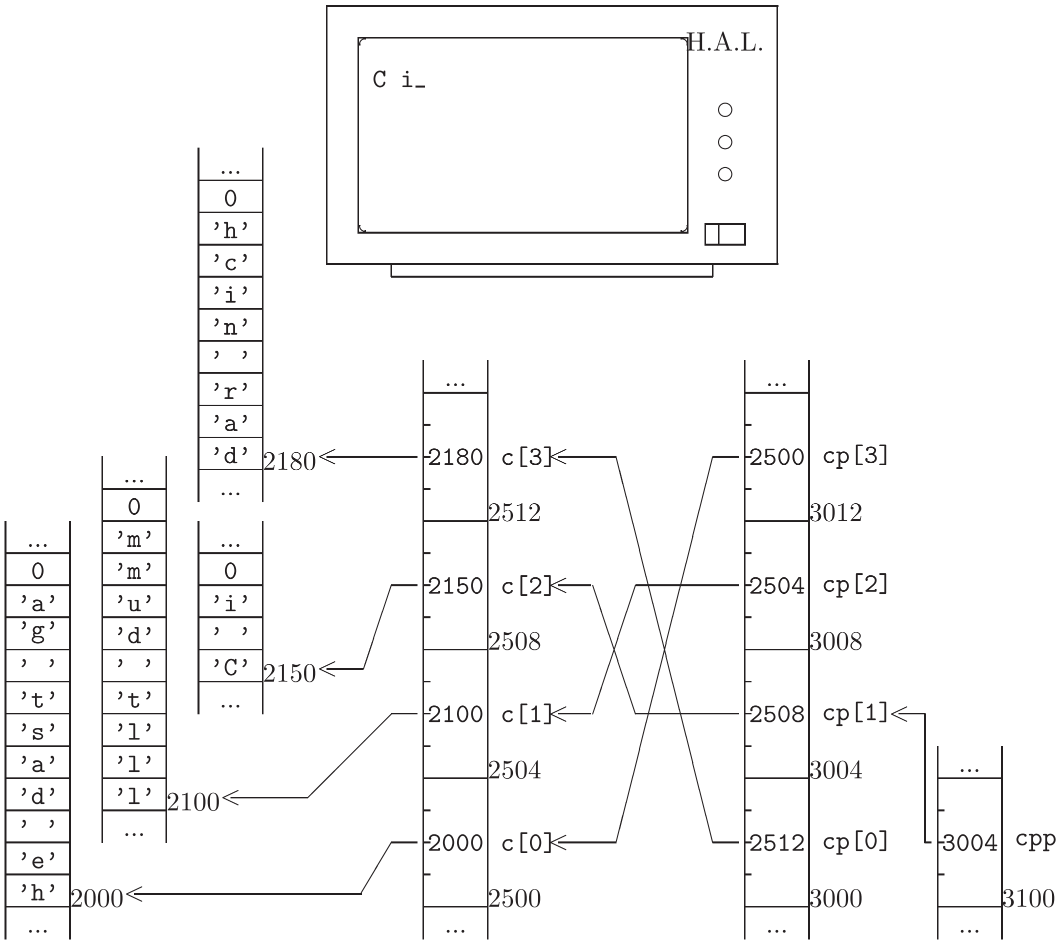
Mit dem rechten * wird der Inhalt von cp[1] beschafft (2508); das ist ein Zeiger auf c[2][36]. Mit dem linken * wird dann wiederum dessen Inhalt beschafft, also der Wert von c[2] (2150). Diese Adresse zeigt auf den Anfang der Stringkonstante "C i", also auf das 'C'. Ab diesem Zeichen wird mit printf() ein String bis zur abschließenden Null ausgegeben, also die drei Zeichen 'C', ' ' und 'i'.

Zu beachten ist aber, daß cpp nun nicht mehr auf cp[0] wie nach der Initialisierung (3000), sondern ab jetzt auf cp[1] (3004) zeigt.

[35] ++ hat eine höhere Priorität als *! Siehe **11 Operatoren und Ausdrücke** auf Seite 83.

[36] Ausdrücke mit dem Operator * werden von rechts her zusammengefaßt, siehe auch Seite 83. Deshalb wird erst der rechte * verwendet, dann der linke.

Unser Beispielspeicher und der Bildschirm haben sich also etwas geändert:



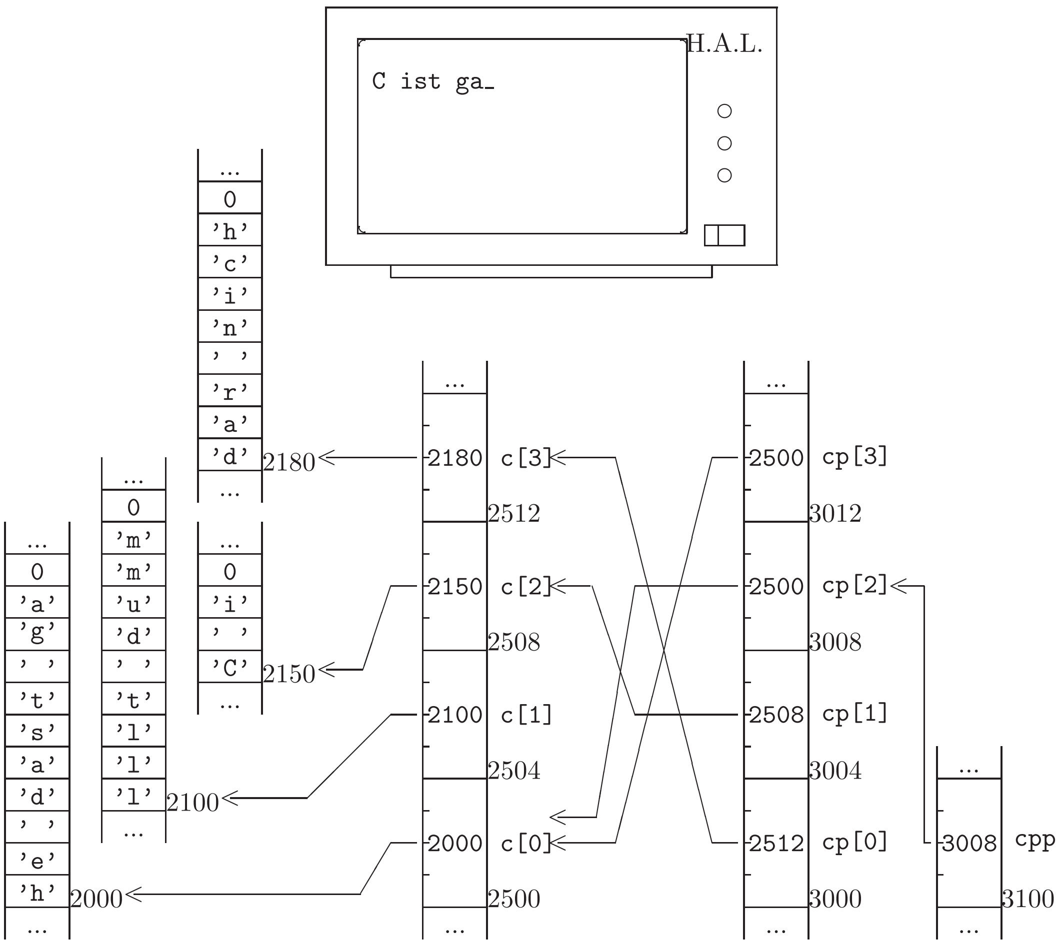
Die Anweisung:

```
printf( "%s", *--*++cpp+5 );
```

erhöht `cpp` nochmals (auf 3008), so daß `cpp` nun auf `cp[2]` zeigt. Mit dem rechten `*` wird nun der Inhalt von `cp[2]` beschafft (2504). Mit `--` wird dieser Wert um 1 erniedrigt (2504-`sizeof(char*)` ergibt 2500), so daß `cp[2]` nicht mehr wie bisher auf `c[1]`, sondern auf `c[0]` zeigt. Mit dem linken `*` wird der Inhalt dieses Zeigers (2000) beschafft; also ein Zeiger auf die erste Stringkonstante.

Der Ausdruck `*--*++cpp` zeigt also auf das erste Zeichen von `"he dast ga"`.

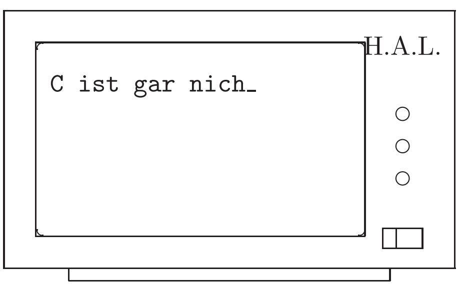
Dementsprechend zeigt `*--*++cpp+5` auf das `s` in der Stringkonstante. Ab diesem Zeichen wird mit `printf()` eine Zeichenkette bis zur abschließenden Null ausgegeben, also die Zeichenkette `st ga`. Damit steht schon der Text `C ist ga` auf dem Bildschirm:



An dem Inhalt der Zeiger ändert sich mit den folgenden Anweisungen nichts mehr: deshalb wird nur noch die Änderung des Bildschirmspeichers gezeigt.

```
printf( "%s", cpp[-2][0]+2 );
```

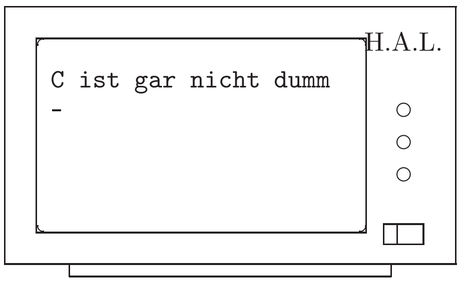
greift mit `cpp[-2]` zwei Elemente vor das, worauf `cpp` zeigt, also auf `cp[0]`, da `cpp` auf `cp[2]` zeigt. Mit `[0]` wird auf das zugegriffen, worauf der zuletzt erhaltene Wert zeigt (`c+3`, bzw. 2180), also ein Zeiger auf die letzte Stringkonstante. `cpp[-2][0]` zeigt also auf das erste Zeichen in `"dar nich"`. `cpp[-2][0]+2` zeigt damit zwei Zeichen dahinter (2182), also auf das `'r'`. Mit dem `printf()`-Aufruf werden folglich die Zeichen `"r nich"` ausgegeben. Jetzt steht insgesamt der Text `"C ist gar nich"` auf dem Bildschirm.



Es geht weiter:

```
printf( "%s\n", *(cpp[1]+1)+3 );
```

greift mit `cpp[1]` auf ein Element zu, das eine Position hinter dem liegt, auf das `cpp` zeigt, also auf `cp[3]` an der Stelle 3012. `cp[3]` enthält den Wert `c` (2500), also die Adresse von `c[0]`. `cpp[1]+1` ist damit die Adresse von `c[1]` (2504). `*(cpp[1]+1)` ist die Adresse des Strings `"lllt dumm"` (2100) und `*(cpp[1]+1)+3` ist die um drei größere Adresse (2103), also die Adresse des `'t'`-Zeichens in dem String. Ab diesem Zeichen wird mit `printf()` wieder bis zur Null ausgegeben, also die Zeichen `"t dumm"`. Insgesamt steht nun der Text `"C ist gar nicht dumm"` auf dem Schirm:




```
getchar();
```

wartet dann nur noch auf eine Eingabe des Benutzers, damit dieser die Programmausgabe in Ruhe lesen kann.

18.2.16 Mit Zeigern jonglieren, aber richtig! (II)

Hier ist meine Musterlösung zu `qsort()`:

```
/* loqsort.c 30. 7.95 kw
 */

#include <stdlib.h>
#include <string.h>

char  *temp;


/* swap() vertauscht base[i] mit base[j], unabhaengig von
 * dem Datentyp der in base[] enthaltenen Elemente.
 */

static void swap( void *base, int i, int  j, size_t len )
{
  char
    *pi = (char*)base + (i*len),
    *pj = (char*)base + (j*len);

  memcpy( temp, pi, len );
  memcpy( pi, pj, len );
  memcpy( pj, temp, len );
}


static void myqsort( void *v,
                     int left,
                     int right,
                     size_t len,
                     int (*compar)(const void*, const void*)
                     )
{
  int i,last;

  if(left >= right)     /* Rekursion abbrechen, wenn nicht    */
    return;             /* mindestens zwei Elemente.          */

  /* Element waehlen und am Vektoranfang sichern:
   */

  swap( v, left,(left + right)/2, len );

  last = left;          /* in last zum Mitzaehlen merken      */

  for(i = left+1; i <= right; i++)    /* aufteilen            */
    if( compar( (char*)v+(i*len), (char*)v+(left*len)) < 0 )
      swap(v, ++last, i, len );

  /* Gewaehltes Element wieder zwischen den Teil mit den
   * kleineren und den Teil mit den groesseren Elementen
```

```
   * setzen:
   */
  swap(v, left, last, len );

  /* linkes und rechtes Teilfeld sortieren:
   */
  myqsort(v,left, last-1, len, compar);
  myqsort(v,last+1,right, len, compar);
}


void qsort( void *base,
            size_t nmemb,
            size_t size,
            int (*compar)(const void*, const void*)
            )
{
  temp = malloc( size );  /* Platz fuer 1 Element             */
  if( temp )
  {
    myqsort((int*)base, 0, 0+nmemb-1, size, compar);
    free( temp );
  }
}
```

18.2.17 Kleiner Praeprozessor

Die Musterlösung wäre etwas zu umfangreich; bei Bedarf bitte auf extra anfordern.

19 Die Standardbibliothek

19.1 Allgemeines

Neben der eigentlichen Sprachbeschreibung legt der ANSI-Standard auch eine Menge von Funktionen mit den zugehörigen Vereinbarungsdateien (Headerdateien) fest, die in jeder *C*-Implementation vorhanden sein müssen.

Diese Funktionen decken einen breiten Bereich von gängigen Teilproblemen ab: Ein- und Ausgabe (Dateien, Benutzer), mathematische Operationen, Sortieren, Suchen, Speicherverwaltung, Stringfunktionen, und so weiter.

Die Funktionen sind so gestaltet, daß sie auf allen üblichen Betriebssystemen implementiert werden können. Spezielle Möglichkeiten einiger Systeme (Verwalten von Unterverzeichnissen, Prozeßsteuerung, parallele Prozesse) sind deshalb im ANSI-Standard nicht zu finden. Da aber die meisten Betriebssysteme über diese Minimalanforderungen hinaus viele Erweiterungen sehr ähnlich handhaben, besitzen die meisten *C*-Implementationen teilweise fast identische Erweiterungen. Ein Überblick, welche Funktion auf den gängigen Systemen zu finden ist, findet sich in [c't port.].

Einige solcher verbreiteten, aber nicht offiziell standardisierten Funktionen, sind hier mit aufgenommen worden. Sie sind als Abweichung vom Standard gekennzeichnet.

Einige der aufgeführten „Funktionen“ können als Makros definiert werden, siehe dazu die Warnungen in **5 Der Praeprozessor** auf Seite 17.

19.2 Fehlerbehandlung

Beim Abarbeiten der Standardfunktionen können natürlich viele Fehler auftreten. Das Verhalten der Funktionen bei etlichen Fehlermöglichkeiten ist im Standard festgelegt. Einige Funktionen signalisieren durch ihren Rückgabewert, daß sie ihren Auftrag nicht ausführen können (beispielsweise liefert `malloc()` den Wert `NULL`, wenn nicht mehr genügend freier Speicher vorhanden ist). Andere Funktionen setzen eine globale Variable `errno`[37], wenn ein Fehler auftritt. Am Wert von `errno` kann man dann die Art des Fehlers erkennen.

Beim Programmstart ist `errno` immer auf 0 gesetzt. Bibliotheksfunktionen setzen den Wert im Fehlerfall; aber nie mehr auf 0. Dadurch muß der Programmierer nicht bei jeder eventuell fehlerhaften Operation `errno` kontrollieren, sondern kann mehrere Funktionen aufrufen und erst dann `errno` prüfen, wenn er es für richtig hält.

Der Programmierer darf mit `errno = 0` den Fehlerindikator löschen.

Die standardisierten Werte von `errno` sind:

`0` solange kein Fehler auftrat.

`EDOM` besagt, daß eine aufgerufene Funktion (aus `math.h`) für das aktuelle Argument mathematisch nicht definiert ist, z.B. `asin(2.0)`. Der Rückgabewert der Funktion ist dann `0.0`.

[37] Nach dem `#include`n der Datei `errno.h` kann man `errno` wie eine als `extern int` vereinbarte Variable verwenden; tatsächlich ist `errno` meistens als Makro mit einem versteckten Funktionsaufruf definiert.

ERANGE tritt auf, wenn das Ergebnis einer Funktion den darstellbaren Zahlenbereich überschreitet (z.B. pow() mit großen Argumenten). Dann wird von der Funktion HUGE_VAL oder -HUGE_VAL zurückgegeben, je nach dem Vorzeichen des (nicht berechenbaren) Ergebnisses.

Andere Fehler werden nicht gemeldet. Beispielsweise darf das Argument einer periodischen Funktion nicht zu groß werden, da sonst durch die begrenzte Stellenzahl der Mantisse die Auflösung innerhalb einer Periode der Funktion nicht mehr reicht. Solche Fehler werden von den Funktionen der Standardbibliothek nicht gemeldet.

Schutz vor Feldgrenzenüberschreitungen (einschließlich Kontrolle von Stringlängen) findet zur Laufzeit nicht statt.

EDOM und ERANGE sind in math.h definiert.

Die Umwandlung von errno-Werten in Text kann man mit den Funktionen **19.4 perror()** auf Seite 209 bzw. **19.4 strerror()** auf Seite 225 erreichen.

Dateien:

Jede geöffnete Datei hat einen sogenannten Fehlerindikator, der beim Auftreten von Fehlern gesetzt wird. Mit ferror() kann man diesen Indikator abfragen.

Beim Öffnen der Datei, beim Zurückspulen auf den Anfang mit rewind() oder beim Aufruf von clearerr() wird der Indikator gelöscht.

Dateifunktionen, die den Typ int als Rückgabewert haben, liefern im Fehlerfall den vordefinierten Wert EOF (definiert in stdio.h).

19.3 Headerdateien und Überblick über die Funktionen

Folgende Headerdateien sind standardisiert:

19.3.1 assert.h

Fehlersuche

Funktionen/funktionsähnliche Makros:

Name	*Zweck*
assert() (S. 178)	Kontrolle von Invarianten in der Debugphase (siehe auch **17 Fehlersuche mit Standard-*C*** auf Seite 130)

Konstanten: -

Datentypen: -

19.3.2 ctype.h

Zeichenklassifizierung und -umwandlung.

Funktionen/funktionsähnliche Makros:

Name	*Zweck*
`isalnum()` (S. 200)	ist das gegebene Zeichen alphanumerisch?
`isalpha()` (S. 200)	ist das gegebene Zeichen ein Buchstabe?
`iscntrl()` (S. 201)	ist das gegebene Zeichen ein Steuerzeichen?
`isdigit()` (S. 201)	ist das gegebene Zeichen eine Ziffer?
`isgraph()` (S. 201)	ist das gegebene Zeichen ein graphisches Zeichen?
`islower()` (S. 202)	ist das gegebene Zeichen ein Kleinbuchstabe?
`isprint()` (S. 202)	ist das gegebene Zeichen druckbar?
`ispunct()` (S. 202)	ist das gegebene Zeichen ein Satzzeichen?
`isspace()` (S. 202)	ist das gegebene Zeichen *white space*?
`isupper()` (S. 203)	ist das gegebene Zeichen ein Großbuchstabe?
`isxdigit()` (S. 203)	ist das gegebene Zeichen eine hexadezimale Ziffer?
`tolower()` (S. 234)	konvertiert ein Zeichen zu einem Kleinbuchstaben
`toupper()` (S. 235)	konvertiert ein Zeichen zu einem Großbuchstaben

Konstanten: -

Datentypen: -

19.3.3 errno.h

globale Variable `errno` und Definitionen zur Fehlerbehandlung, siehe **19.2 Fehlerbehandlung** auf Seite 166.

Funktionen/funktionsähnliche Makros: -

Konstanten: -

Datentypen: -

19.3.4 float.h

Definitionen für Wertebereiche der Typen `float`, `double` und `long double`.

Siehe **8.4.3 Gebrochene Zahlen** auf Seite 40.

Funktionen/funktionsähnliche Makros: -

Konstanten:

- `FLT_`...
- `DBL_`...

Datentypen: -

19.3.5 limits.h

Definitionen für Wertebereiche der ganzzahligen Datentypen.

Siehe **8.4.1 Ganzzahlige Datentypen** auf Seite 38.

Funktionen/funktionsähnliche Makros: -

Konstanten:

- Kleinst- und Größtwerte für ganzzahlige Datentypen, wie beispielsweise `INT_MAX`

Datentypen: -

19.3.6 locale.h

Definitionen zur Einstellung landes- und sprachspezifischer Eigenschaften der Bibliotheksfunktionen. Die meisten Implementationen unterstützen keine weiteren Einstellungen, deshalb meist bedeutungslos.

Funktionen/funktionsähnliche Makros: -

Konstanten: -

Datentypen: -

19.3.7 math.h

Funktionsdeklarationen der mathematischen Funktionen.

Definition der Makros `EDOM`, `ERANGE` und `HUGE_VAL`.

Funktionen/funktionsähnliche Makros:

Name	*Zweck*
`acos()` (S. 177)	Arkuskosinus
`asin()` (S. 178)	Arkussinus
`atan()` (S. 178)	Arkustangens (2 Quadranten)
`atan2()` (S. 179)	Arkustangens (4 Quadranten)
`ceil()` (S. 184)	Gleitkommazahl aufrunden
`cos()` (S. 186)	Kosinus
`cosh()` (S. 186)	Kosinus hyperbolicus
`exp()` (S. 188)	Exponentialfunktion (e^x)
`fabs()` (S. 188)	Absolutwert einer Gleitkommazahl
`floor()` (S. 191)	Abrunden
`fmod()` (S. 192)	Divisionsrest bei Gleitkommazahlen
`frexp()` (S. 196)	Zerlegen einer `double` in Mantisse und Binärexponent
`ldexp()` (S. 203)	Multiplikation einer `double` mit Zweierpotenz
`log()` (S. 204)	natürlicher Logarithmus (zur Basis e)
`log10()` (S. 205)	dekadischer Logarithmus (zur Basis 10)
`modf()` (S. 208)	Zerlegen einer `double` in Vor- und Nachkommateil
`pow()` (S. 209)	Potenzieren (siehe auch **A.2 Potenzen mit ganzzahligen Exponenten** auf Seite 245)
`sin()` (S. 222)	Sinus
`sinh()` (S. 223)	Sinus hyperbolicus
`sqrt()` (S. 223)	Quadratwurzel
`tan()` (S. 232)	Tangens
`tanh()` (S. 232)	Tangens hyperbolicus

Konstanten:

- `errno`-Werte `EDOM` (*domain error*), `ERANGE` (*range error*) und `HUGE_VAL` (*„riesiger Wert“*)

Datentypen: -

19.3.8 setjmp.h

Funktionsdeklarationen zu `setjmp()` und `longjmp()` zum Sprung über Funktionsgrenzen hinweg. Siehe **longjmp()** (Seite 205).

Funktionen/funktionsähnliche Makros:

Name	*Zweck*
`longjmp()` (S. 205)	Sprung zu einem mit `setjmp()` eingefrorenen Stand
`setjmp()` (S. 221)	Sichern eines Programmzustands für `longjmp()`

Konstanten: -

Datentypen:

- `jmp_buf` kann einen mit `setjmp()` eingefrorenen Registersatz aufnehmen

19.3.9 signal.h

Handhabung von Signalen.

Funktionen/funktionsähnliche Makros:

Name	*Zweck*
`raise()` (S. 215)	Herbeiführen eines Signals für den aktuellen Prozeß
`signal()` (S. 221)	Einstellen der Reaktion auf ein Signal

Konstanten:

- `SIGABRT` Programmabbruch, z.B. wegen Aufruf von `abort()`.
- `SIGFPE` Fehler bei Gleitkommarechnung.
- `SIGILL` unzulässiger Funktionsaufruf.
- `SIGINT` Eintreffen eines interaktiven Signals.
- `SIGSEGV` Unzulässiger Speicherzugriff.
- `SIGTERM` Programmende gefordert.
- *sowie je nach System noch weitere. . .*

Datentypen: -

19.3.10 stdarg.h

Verwendung variabler Parameterlisten, siehe **19.4 va_start(), va_arg(), va_end()** auf Seite 235.

Mit den in stdarg.h definierten Makros ist es innerhalb einer Funktion mit variabler Argumentliste möglich, die vom Aufrufer übergebenen Parameter zu erreichen.

Funktionen/funktionsähnliche Makros:

Name	*Zweck*
`va_start()` (Seite 235)	Initialisierung zum Holen der Parameter
`va_arg()` (Seite 235)	je Aufruf einen Parameter holen
`va_end()` (Seite 235)	Beenden

Konstanten: -

Datentypen:

- va_list (Zeiger in den Stack auf das jeweils nächste zu holende Argument)

.

19.3.11 stddef.h

Allgemein benötigte Definitionen

Funktionen/funktionsähnliche Makros: -

Konstanten:

- NULL („Ungültiger Zeiger“)

Datentypen:

- size_t für Größenangaben
- ptrdiff_t für die Differenz zweier Adressen

19.3.12 stdio.h

Funktionen und Makros zur gepufferten Ein- und Ausgabe

Funktionen/funktionsähnliche Makros:

Name	*Zweck*
`clearerr()` (S. 185)	
`fclose()` (S. 188)	Schließen einer Datei
`feof()` (S. 188)	Prüfen auf Dateiende
`ferror()` (S. 189)	Prüfen auf Fehlerzustand
`fflush()` (S. 189)	Zurückschreiben gepufferter Daten
`fgetc()` (S. 189)	Lesen eines Zeichens
`fgetpos()` (S. 190)	Abfragen der Dateiposition zum Setzen mit `fsetpos()`
`fgets()` (S. 190)	Lesen eine Zeile
`fopen()` (S. 192)	Öffnen einer Datei
`fprintf()` (S. 193)	formatierte Ausgabe
`fputc()` (S. 194)	Schreiben eines Zeichens
`fputs()` (S. 194)	Schreiben eines Strings
`fread()` (S. 194)	Lesen eines Blocks
`freopen()` (S. 195)	nachträgliches Ändern von Attributen
`fscanf()` (S. 196)	formatiertes Lesen
`fseek()` (S. 197)	Positionieren (relativ oder absolut)
`fsetpos()` (S. 198)	Positionieren auf eine von `fgetpos()`-Position
`ftell()` (S. 198)	Abfragen der Dateiposition zum Setzen mit `fseek()`
`fwrite()` (S. 198)	Schreiben eines Blocks
`getc()` (S. 199)	Lesen eines Zeichens
`getchar()` (S. 199)	Lesen eines Zeichens (von `stdin`)
`gets()` (S. 199)	Lesen eine Zeile (von `stdin`)
`perror()` (S. 209)	Ausgabe einer Fehlermeldung
`printf()` (S. 210)	formatierte Ausgabe (nach `stdout`)
`putc()` (S. 212)	Ausgabe eines Zeichens
`putchar()` (S. 212)	Ausgabe eines Zeichens (nach `stdout`)
`puts()` (S. 213)	Schreiben eines Strings (nach `stdout`)
`remove()` (S. 217)	Löschen einer Datei
`rename()` (S. 217)	Umbenennen einer Datei
`rewind()` (S. 217)	An den Anfang spulen
`scanf()` (S. 218)	formatiertes Lesen (von `stdin`)
`setbuf()` (S. 220)	Vorgabe eines Puffers
`setvbuf()` (S. 221)	feinere Einstellung der Pufferung
`sprintf()` (S. 223)	formatierte Ausgabe in ein `char`-Feld
`sscanf()` (S. 224)	formatiertes Lesen aus einem String
`tmpfile()` (S. 234)	Öffnen einer temporären Datei
`tmpnam()` (S. 234)	Bilden eines eindeutigen Namens für temporäre Datei
`ungetc()` (S. 235)	Zurückstellen eines gelesenen Zeichens
`vfprintf()` (S. 238)	Ausgabe einer variablen Parameterliste
`vsprintf()` (S. 238)	Ausgabe einer variablen Parameterliste in `char`-Feld
`vprintf()` (S. 239)	Ausgabe einer variablen Parameterliste `stdout`

Datentypen:

- `FILE` (intern verwendete Beschreibung jeder geöffneten Datei)
- `fpos_t` (Dateiposition für **fgetpos()** (Seite 190) und **fsetpos()** (Seite 198))

Konstanten:

- EOF (für einige Funktionen Rückgabewert im Fehlerfall; siehe **15 Dateien** auf Seite 109)
- Die vordefinierten Dateizeiger (vom Typ FILE *) stdin (meist Tastatur), stdout (meist Bildschirm), stderr (meist Bildschirm für Fehlermeldungen).
- Makros für die möglichen Werte des letzten Parameters der Funktion **fseek()** (Seite 197): SEEK_SET, SEEK_CUR und SEEK_END.

  Makros für die Art der Dateipufferung mit **setvbuf()** (Seite 221): _IONBF, _IOLBF, _IOFBF.

19.3.13 stdlib.h

Deklarationen aller Funktionen, die nicht zu anderen Funktionsgruppen passen (wie etwa malloc(), qsort(), . . .).

Funktionen/funktionsähnliche Makros:

Name	*Zweck*
abort() (S. 176)	Programmabbruch
abs() (S. 176)	Absolutwert einer ganzen Zahl
atexit() (S. 179)	Vormerken einer bei Programmende auszuführenden Funktion
atof() (S. 180)	Gleitkommazahl aus String lesen (veraltet)
atoi() (S. 180)	int aus String lesen (veraltet)
atol() (S. 181)	long aus String lesen (veraltet)
bsearch() (S. 181)	binäre Suche in sortiertem Feld beliebiger Elemente; freie Definition des Sortierkriteriums
calloc() (S. 184)	Speicher allokieren und zu 0 setzen
div() (S. 187)	Quotient und Divisionsrest int/int
exit() (S. 187)	Programm beenden
free() (S. 195)	allokierten Speicher freigeben
getenv() (S. 199)	Umgebungsvariable abfragen
labs() (S. 203)	Absolutbetrag einer long
ldiv() (S. 204)	Quotient und Divisionsrest long/long
malloc() (S. 206)	Speicher allokieren
qsort() (S. 213)	Sortieren eines Felds beliebiger Elemente; freie Definition des Sortierkriteriums
rand() (S. 215)	Berechnen einer (Pseudo-)Zufallszahl
realloc() (S. 216)	Speicher allokieren, freigegeben, allokierte Größe ändern
srand() (S. 223)	Startwert für rand() setzen
strtod() (S. 230)	Gleitkommazahl aus String lesen
strtol() (S. 231)	vorzeichenbehaftete ganze Zahl aus String lesen
strtoul() (S. 231)	vorzeichenlose ganze Zahl aus String lesen
system() (S. 232)	Aufruf eines Kommandointerpreters oder eines anderen Programms

Konstanten:

- EXIT_SUCCESS (Argument für `exit()` bzw. Rückgabewert für `main()` bei erfolgreichem Programmlauf)
- EXIT_FAILURE (dto. bei fehlerhaftem Programmlauf)

Datentypen: -

19.3.14 string.h

Handhabung von Strings und Speicherbereichen.

Funktionen, deren Namen mit `mem`... beginnen, werden für Speicherbereiche fester Länge verwendet.

Funktionen, die mit `str`... beginnen, arbeiten mit normalen *zero terminated strings* (nullterminierten Zeichenketten); siehe **8.8.2 Strings** auf Seite 69. Funktionen mit `strn`... am Namensanfang ebenso, aber mit maximaler Länge.

string.h heißt auf älteren Unix-Systemen manchmal strings.h.

Funktionen/funktionsähnliche Makros:

Name	*Zweck*
`memchr()` (S. 207)	Zeichen in Speicher suchen
`memcmp()` (S. 207)	zwei Speicherbereiche vergleichen
`memcpy()` (S. 207)	Speicherbereich kopieren
`memmove()` (S. 208)	Speicherbereich kopieren (ggf. mit Überlappung)
`memset()` (S. 208)	Speicherbereich füllen
`strcat()` (S. 224)	Strings aneinanderhängen
`strchr()` (S. 224)	in String nach einem Zeichen suchen
`strcmp()` (S. 224)	zwei Strings vergleichen
`strcpy()` (S. 225)	String kopieren
`strcspn()` (S. 225)	Stringanfang untersuchen (solange nicht gegebene Zeichen)
`strerror()` (S. 225)	Fehlermeldung formatieren anhand `errno`-Wert
`strlen()` (S. 227)	Stringlänge bestimmen
`strncat()` (S. 228)	Strings aneinanderhängen
`strncmp()` (S. 228)	zwei Strings vergleichen
`strncpy()` (S. 228)	String kopieren
`strpbrk()` (S. 229)	in String nach einem von mehreren Zeichen suchen
`strrchr()` (S. 229)	in String (von hinten) nach einem Zeichen suchen
`strspn()` (S. 229)	Stringanfang untersuchen (solange gegebene Zeichen)
`strstr()` (S. 229)	In String nach String suchen
`strtok()` (S. 230)	String in Token zerlegen

Konstanten: -

Datentypen: -

19.3.15 time.h

Datum und Uhrzeit, Rechenzeit

Zur Laufzeitmessung eines Programms gibt es die Funktion `clock()`, welche die Anzahl der „Schläge" der rechnerinternen Uhr (seit Programmstart) liefert. Das Makro CLOCKS_PER_SEC gibt die Anzahl dieser Schläge pro Sekunde an und kann zum Umrechnen in Sekunden benutzt werden. Der Ergebnistyp von `clock()` heißt `clock_t` und stellt in der Regel eine ganze Zahl dar, die auf Sekunden umgerechnet werden kann.

Beispiel siehe **19.4 clock()** auf Seite 185.

Funktionen/funktionsähnliche Makros:

Name	*Zweck*
`asctime()` (S. 177)	liefert Stringdarstellung einer `struct tm`
`clock()` (S. 185)	liefert Anzahl Timerticks (verbrauchte CPU-Zeit) seit Programmstart
`ctime()` (S. 187)	liefert Stringdarstellung einer `time_t`
`difftime()` (S. 187)	Differenz zweier `time_t`
`gmtime()` (S. 200)	berechnet Greenwich Mean Time als `struct tm` aus der Ortszeit in `time_t`
`localtime()` (S. 204)	berechnet Ortszeit als `struct tm` aus der Ortszeit in `time_t`
`mktime()` (S. 208)	berechnet aus Ortszeit als `struct tm` die gepackte Form `time_t`
`strftime()` (S. 226)	formatiertes Schreiben einer Zeit (`struct tm`) in String
`time()` (S. 232)	aktuelle Systemzeit als `time_t`

Konstanten: -

Datentypen: Datum und Uhrzeit werden in zwei verschiedenen Formen dargestellt.

- `time_t` ist eine gepackte, bitweise organisierte Struktur, deren genauer Aufbau systemabhängig und nicht von Bedeutung ist.
- `struct tm` dagegen ist portabel und hat (mindestens) folgende Einträge:

```
struct tm
{
    int    tm_sec;    /* Sekunde              0...59     */
    int    tm_min;    /* Minute               0...59     */
    int    tm_hour;   /* Stunde               0...23     */
    int    tm_mday;   /* Monatstag            1...31     */
    int    tm_mon;    /* Monat                0...11     */
    int    tm_year;   /* Jahreszahl - 1900               */
    int    tm_wday;   /* Wochentag So==0                 */
    int    tm_yday;   /* Tag im Jahr          0...365    */
    int    tm_isdst;  /* Sommerzeit==1                   */
};
```

Nicht standardisiert, aber relativ weit verbreitet sind die Headerdateien conio.h (ungepufferte Ein-/Ausgabe mit Terminal), io.h und fcntl.h (ungepufferte Dateien), process.h (Starten weiterer Prozesse) sowie direct.h (Verzeichnisverwaltung).

19.4 Liste der Funktionen

In diesem Abschnitt werden die Funktionen aus der Standardbibliothek von *ANSI-C* aufgelistet.

Nach der Angabe der `#include`-Datei, in der die jeweilige Funktion deklariert ist, und der Deklaration selbst, wird die Funktion vorgestellt. Soweit vorhanden, werden Parameter und Rückgabewert angegeben.

Von Fall zu Fall wird ein kleines Verwendungsbeispiel angegeben.

Die Funktionen sind alphabetisch sortiert.

Alle Funktionen, die in einer tatsächlichen Implementation vorhanden sind, aber hier nicht oder mit einer anderen Funktionsweise genannt sind, sollten so weit als möglich vermieden werden, da der Quelltext dann nicht mehr portierbar ist.

Müssen solche Funktionen unbedingt verwendet werden, sollte man darauf achten, daß sie nur in wenigen, möglichst übersichtlichen Teilen des Programms vorkommen und von der eigentlichen Programmlogik möglichst weit getrennt werden. Das Abtrennen vom eigentlichen Programm und deutliches Kommentieren aller Abweichungen vom ANSI-Standard erleichtert ein späteres Übertragen sowie Einarbeiten fremder Personen enorm. Für spätere Portierungen ist es außerdem sehr hilfreich, wenn bei zusätzlichen Funktionen, die nicht standardisiert sind, als Kommentar die genaue Funktionsweise oder eine entsprechende Literaturangabe im Quelltext erscheint. Sonst kann es passieren, daß die Funktion auf einem anderen System nicht existiert und nachgebildet werden muß, aber niemand mehr so genau weiß, was die Funktion eigentlich macht.

abort()

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

```
void abort( void );
```

bricht das Programm unmittelbar ab.

Funktionen, die mit `atexit()` eingetragen wurden, werden nicht ausgeführt; offene Dateien werden nicht geschlossen oder ihre Puffer ausgeschrieben; temporäre Dateien (mit `tmpfile()` geöffnet) werden nicht gelöscht. Das Signal `SIGABRT` wird mit `raise( SIGABRT )` erzeugt und das Programm dann mit dem Rückgabewert `3` abgebrochen.

→**exit()** (Seite 187), **raise()** (Seite 215), **assert()** (Seite 178)

abs()

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

```
int abs( int wert );
```

gibt den Absolutbetrag von `wert` zurück.

→**labs()** (Seite 203), **fabs()** (Seite 188)

acos()

```
#include <math.h>

double acos( double argument );
```

liefert den ersten Wert des arccos(`argument`) im Bogenmaß (im Bereich 0.0...π). `argument` muß dazu im Bereich -1.0...+1.0 liegen. Ansonsten wird 0.0 zurückgegeben und `errno` wird auf `EDOM` gesetzt.

→**cos()** (Seite 186), **asin()** (Seite 178), **atan()** (Seite 178), **atan2()** (Seite 179)

asctime()

```
#include <time.h>

char *asctime( const struct tm *tm_zeiger );
```

liefert einen Zeiger auf einen String, der die Zeit in `*tm_zeiger` beschreibt. Beispielsweise liefert das Programm

```
#include <time.h>
#include <stddef.h>             /* wegen NULL                */

main()
{
    time_t      t;

    t = time( NULL );       /* Die Uhrzeit des Rechners lesen */

    /* Die aktuelle Zeit in eine struct tm verwandeln,
     * von da in einen String und ausgeben
     */
    printf( "es ist jetzt %s",
            asctime( localtime( &t ) ) );
    printf( "ok.\n" );
}
```

die aktuelle Uhrzeit in der Form

```
es ist jetzt Mon Mar 11 20:37:20 1991
ok.
```

Eine für hiesige Gewohnheiten gefälligere Form der Zeitdarstellung wird auf Seite 232 vorgeführt.

Die Positionen der einzelnen Felder in dem 26 Zeichen langen String sind immer gleich und der String ist mit `\n` und `\0` beendet.

Platz für den String braucht nicht bereitgestellt zu werden, sondern ist statisch in der Funktion `asctime()` vorhanden und wird beim nächsten Aufruf derselben oder `ctime()` überschrieben.

→**ctime()** (Seite 187), **strftime()** (Seite 226), **mktime()** (Seite 208), **time()** (Seite 232)

asin()

```
#include <math.h>

double asin( double argument );
```

liefert den ersten Wert des arcsin(`argument`) im Bogenmaß (im Bereich $-\pi/2 \ldots +\pi/2$). `argument` muß dazu im Bereich -1.0. . . +1.0 liegen. Ansonsten wird 0.0 zurückgegeben und `errno` wird auf `EDOM` gesetzt.

→**sin()** (Seite 222), **acos()** (Seite 177), **atan()** (Seite 178), **atan2()** (Seite 179)

assert()

```
#include <assert.h>

void assert( int ausdruck );
```

ist ein Makro zur Fehlersuche während der Programmentwicklung.

Wenn vor der Praeprozessoranweisung `#include <assert.h>` die Anweisung

```
#define NDEBUG
```

steht, dann wird `assert()` vom Praeprozessor aus dem Programm entfernt und bewirkt nichts.

Ansonsten wird vom Praeprozessor ein Stück Quelltext so erzeugt, daß der `ausdruck` ausgewertet wird und im Falle von 0 bzw. *logisch falsch* wird eine Meldung in der Form

`assertion` *ausdruck* `failed: file` *Dateiname* `line` *Zeilennummer*

ausgegeben und anschließend das Programm mit `abort()` abgebrochen.

Siehe auch die Funktion `abort()`.

Ein Beispiel zur Verwendung von `assert()` ist in **17 Fehlersuche mit Standard-*C*** auf Seite 130 zu finden.

→**abort()** (Seite 176), **exit()** (Seite 187)

atan()

```
#include <math.h>

double atan( double argument );
```

liefert den ersten Wert von arctan(`argument`) im Bogenmaß (im Bereich $-\frac{\pi}{2} \ldots +\frac{\pi}{2}$).

→**tan()** (Seite 232), **asin()** (Seite 178), **acos()** (Seite 177), **atan2()** (Seite 179)

atan2()

```
#include <math.h>

double atan2( double zaehler, double nenner );
```

liefert den ersten Wert des $\arctan(\frac{\texttt{zaehler}}{\texttt{nenner}})$ im Bogenmaß (im Bereich $-\pi\ldots+\pi$).

zaehler und nenner dürfen dabei nicht beide den Wert 0.0 haben; in diesem Fall wird errno auf EDOM gesetzt und 0.0 zurückgegeben.

→**tan()** (Seite 232), **asin()** (Seite 178), **acos()** (Seite 177), **atan()** (Seite 178)

atexit()

```
#include <stdlib.h>

int atexit( void ( *funktion_zeiger )( void );
```

sorgt dafür, daß bei einem normalen Programmende (return in main() oder exit()) die Funktion, auf die funktion_zeiger zeigt, aufgerufen wird. *funktion_zeiger erhält keinen Parameter und liefert keinen Rückgabewert. Werden mehrere Funktionen so installiert, dann werden sie bei Programmende entgegen ihrer Installationsreihenfolge ausgeführt. Zu diesem Zeitpunkt existieren keine automatischen Variablen mehr, aber offene Dateien und temporäre Dateien sind noch verwendbar.

Wird das Programm mit abort() beendet, dann werden die mit atexit() installierten Funktionen nicht ausgeführt.

Rückgabewert ist ungleich 0, wenn zuviele Einträge mit atexit() installiert wurden.

Beispielaufruf:

```
/* fuassert.c 30. 7.95 kw
 */

#include <stdlib.h>

void erste( void )
{
  printf( "Hallo, hier ist die erste!\n" );
}

void zweite( void )
{
  printf( "Hallo, hier ist die zweite!\n" );
}

int main()
{
  /* die beiden obigen Funktionen installieren,
   * damit sie bei Programmende aufgerufen werden:
    */
  atexit( &erste );
  atexit( &zweite );
```

```
  /* Jetzt koennte das Programm irgendetwas tun ...
   */
  printf( "irgendetwas tun ...\n" );

  /* ... und dann beenden.
   * Nach diesem return werden die mit atexit()
   * installierten Funktionen in umgekehrter
   * Reihenfolge ausgefuehrt:
   */
  return 0;
}
```

Ausgabe davon:

```
irgendetwas tun ...
Hallo, hier ist die zweite!
Hallo, hier ist die erste!
```

→**exit()** (Seite 187), **abort()** (Seite 176)

atof()

```
#include <stdlib.h>

double atof( const char *string );
```

verwandelt `string` soweit möglich in eine `double`; dabei wird beim ersten Zeichen in `string` abgebrochen, das nicht als Bestandteil einer Gleitkommazahl interpretiert werden kann. Kann überhaupt kein Zeichen umgewandelt werden, dann wird 0.0 zurückgegeben.

Die Funktion `atof()` sollte für Neuentwicklungen nicht mehr verwendet werden. Sie ist nur zur Kompatibilität mit älteren *C*-Versionen vorhanden und kann durch `strtod( string, NULL )` ersetzt werden.

Wenn der Wertebereich von `double` nicht ausreicht, um das Ergebnis der Umwandlung darzustellen, dann ist der Rückgabewert unbestimmt.

→**strtod()** (Seite 230), **atoi()** (Seite 180), **atol()** (Seite 181), **sscanf()** (Seite 224)

atoi()

```
#include <stdlib.h>

int atoi( const char *string );
```

verwandelt `string` soweit möglich in eine `int`; dabei wird beim ersten Zeichen in `string` abgebrochen, das nicht als Bestandteil einer ganzen Zahl interpretiert werden kann. Kann überhaupt kein Zeichen umgewandelt werden, dann wird 0 zurückgegeben.

Die Funktion `atoi()` sollte für Neuentwicklungen nicht mehr verwendet werden. Sie ist nur zur Kompatibilität mit älteren *C*-Versionen vorhanden und kann durch

```
(int)strtol( string, NULL, 10 )
```

ersetzt werden.

Wenn der Wertebereich von `int` nicht ausreicht, um das Ergebnis der Umwandlung darzustellen, dann ist der Rückgabewert unbestimmt.

→**strtol()** (Seite 231), **atof()** (Seite 180), **atol()** (Seite 181), **sscanf()** (Seite 224)

atol()

```
#include <stdlib.h>

long int atol( const char *string );
```

verwandelt `string` soweit möglich in eine `long int`; dabei wird beim ersten Zeichen in `string` abgebrochen, das nicht als Bestandteil einer ganzen Zahl interpretiert werden kann. Kann überhaupt kein Zeichen umgewandelt werden, dann wird `0L` zurückgegeben.

Die Funktion `atol()` sollte für Neuentwicklungen nicht mehr verwendet werden. Sie ist nur zur Kompatibilität mit älteren *C*-Versionen vorhanden und kann durch `strtol( string, NULL, 10 )` ersetzt werden.

Wenn der String eine zu große Zahl für den Wertebereich von `long int` ergibt, dann ist das Ergebnis nicht definiert und `errno` wird zu `ERANGE` gesetzt.

→**atoi()** (Seite 180), **atof()** (Seite 180), **sscanf()** (Seite 224)

bsearch()

```
#include <stdlib.h>

void *bsearch( const void *suchschluessel,
               const void *feldanfang,
               size_t     elemente,
               size_t     elementgroesse,
               int ( *vergleichsfunktion )( const void *e1,
                                            const void *e2  )
             );
```

sucht in einem Feld nach einem Element, das den Wert `(*suchschluessel)` hat und liefert einen Zeiger auf dieses Element, wenn folgende Bedingungen erfüllt sind:

- Das Feld beginnt an der Stelle `feldanfang` im Speicher.
- Das Feld hat `(elemente)` Elemente jeweils der Größe `(elementgroesse)`.
- `vergleichsfunktion` zeigt auf eine Funktion, die zwei Zeiger auf je ein Feldelement entgegennimmt und einen negativen `int`-Wert liefert, wenn der erste Parameter auf ein Element mit kleinerem Suchschlüssel zeigt als der zweite Parameter, 0 liefert, wenn beide Parameter auf Elemente mit gleich großem Suchschlüssel zeigen und sonst einen positiven Wert liefert.
- Das Feld ist gemäß den Kriterien der `vergleichsfunktion` aufsteigend sortiert.

Trifft `(*suchschluessel)` auf keines der Elemente im Feld zu, dann liefert `bsearch()` den Wert `NULL`.

Trifft `(*suchschluessel)` auf mehrere Feldelemente zu, dann wird eines davon geliefert; aber es kann nicht vorhergesagt werden, welches.

Das folgende Beispiel liest einige Vor- und Nachnamen ein und sortiert sie nach den Vornamen. Dann kann der Benutzer einen Vornamen eingeben, zu dem der Nachname gesucht wird:

```
/* fubsear.c 30. 7.95 kw
 */

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

#define         NP      5

/* Datentyp, um einen Vor- und einen Nachnamen zu speichern:
 */
typedef struct
{
  char    vorname[30];
  char    nachname[30];
}
person_t;

/* davon ein Feld:
 */
person_t        personfeld[NP];

/* vergleicht fuer qsort() oder bsearch() zwei Feldelemente
 * (nach Vornamen):
 */
int vergl_vn( const void *p1, const void *p2 )
{
  return strcmp( ((person_t *)p1)->vorname,
                 ((person_t *)p2)->vorname
                 );
}

int main()
{
  int         i;
  person_t
    vergleichsperson,
    *gefunden;

  /* Personen einlesen:
   */
  for( i=0; i<NP; i++ )
  {
    printf( "Bitte den Vornamen:  " );
    scanf( "%s", personfeld[i].vorname );
```

```
    printf( "Bitte den Nachnamen: " );
    scanf( "%s", personfeld[i].nachname );
  }

  /* Nach Vornamen sortieren:
   */
  qsort( personfeld, NP, sizeof(person_t), vergl_vn );

  /* Nach einem Vornamen fragen und die Person suchen:
   */
  printf( "Bitte einen Vornamen:  " );
  scanf( "%s", vergleichsperson.vorname );
  gefunden = bsearch( &vergleichsperson,
                      personfeld,
                      NP,
                      sizeof(person_t),
                      vergl_vn
                      );
  if( NULL==gefunden )
  {
    puts( "Den kann ich nicht finden!" );
  }
  else
  {
    printf( "%s %s\n",
            gefunden->vorname,
            gefunden->nachname );
  }

  return 0;
}
```

Bei älteren, nicht ANSI-konformen Compilern kann es sein, daß die Anweisung

```
        gefunden = bsearch( &vergleichsperson,
                            personfeld,
                            NP,
                            sizeof(person_t),
                            vergl_vn             );
```

geändert werden muß zu:

```
        gefunden = bsearch( &vergleichsperson,
                            personfeld,
                            NP,
                            sizeof(person_t),
                            &vergl_vn            );
```

(siehe dazu **Zeiger auf Funktionen** (Seite 44)).

→**qsort()** (Seite 213)

calloc()

```
#include <stdlib.h>

void *calloc( size_t wieviele, size_t wiegross );
```

versucht, einen Speicherblock mit `(wieviele*wiegross)` Bytes zu reservieren. Der reservierte Speicher wird in ganzer Länge auf den Wert 0 gesetzt.

Rückgabewert ist ein Zeiger auf den Anfang des reservierten Blocks, wenn alles gut ging, oder der `NULL`-Zeiger, wenn kein Block ausreichender Größe zu haben ist.

→**free()** (Seite 195), **malloc()** (Seite 206), **realloc()** (Seite 216)

ceil()

```
#include <math.h>

double ceil( double wert );
```

liefert die nächstgrößere ganze Zahl zu `wert` als `double`.

Beispiel:

```
#include <math.h>

int main()
{
  double  d;

  for( d=-2.0; d<2.0; d+=0.2 )
    printf( "ceil(%f) = %f\n", d, ceil(d) );
}
```

Dieses Programm liefert die Ausgabe:

```
ceil(-2.000000) = -2.000000
ceil(-1.800000) = -1.000000
ceil(-1.600000) = -1.000000
ceil(-1.400000) = -1.000000
ceil(-1.200000) = -1.000000
ceil(-1.000000) = -1.000000
ceil(-0.800000) = -0.000000
ceil(-0.600000) = -0.000000
ceil(-0.400000) = -0.000000
ceil(-0.200000) = -0.000000
ceil(0.000000) = 1.000000
ceil(0.200000) = 1.000000
ceil(0.400000) = 1.000000
ceil(0.600000) = 1.000000
ceil(0.800000) = 1.000000
ceil(1.000000) = 2.000000
ceil(1.200000) = 2.000000
ceil(1.400000) = 2.000000
ceil(1.600000) = 2.000000
ceil(1.800000) = 2.000000
```

→**floor()** (Seite 191)

clearerr()

```
#include <stdio.h>

void clearerr( FILE *datei );
```

löscht gegebenenfalls die Indikatoren für Dateifehler und Dateiende für die übergebene Datei.

`clearerr()` kann auch als Makro definiert sein.

→**feof()** (Seite 188), **ferror()** (Seite 189), **fopen()** (Seite 192), **freopen()** (Seite 195), **fclose()** (Seite 188), **rewind()** (Seite 217)

clock()

```
#include <time.h>

clock_t clock( void );
```

liefert die Anzahl der „Schläge" der internen Rechneruhr seit Programmbeginn.

Dies ist ein Maß für die Zeit, die das Programm bereits im Rechner läuft.

Die Anzahl der Uhrschläge pro Sekunde ist mit dem Makro `CLOCKS_PER_SEC` definiert. Zur Umrechnung des Funktionsergebnisses von `clock()` in Sekunden kann man das Funktionsergebnis deshalb durch `CLOCKS_PER_SEC` teilen[38].

Je nach Rechnersystem kann man nicht sicher sein, daß diese Funktion funktioniert, da die Zeitinformation nicht immer zur Verfügung steht. Dann liefert `clock()` immer den Wert -1.

Beispiel:

```
/* fuclock.c 30. 7.95 kw
 */

#include <time.h>
#include <stdio.h>

int main()
{
  clock_t
    start,
    ende;
  long int  i;
  char      name[30];
```

[38] `CLOCKS_PER_SEC` hieß in einem vorläufigen Entwurf der Normung `CLK_TCK`; diese Bezeichnung ist bei manchen Compilern noch anzutreffen. Um mit einem solchen Compiler aber portable Programme erstellen zu können, sollte man in diesem Fall trotzdem `CLOCKS_PER_SEC` verwenden und die Headerdatei `time.h` durch Anfügen von

```
#define CLOCKS_PER_SEC CLK_TCK
```

auf den neuesten Stand bringen.

```
  start = clock();

  if( -1==start )
  {
    puts( "Ich kann die Zeit nicht messen!" );
  }
  else
  {
    printf( "Bitte Ihren Namen eingeben: " );
    scanf( "%s", name );
    ende = clock();
    printf( "Sie brauchten %f sec fuer "
            "Ihren Namen, Sie %s!\n",
            (double)(ende-start)/CLOCKS_PER_SEC,
            name
            );
  }

  return 0;
}
```

→**time()** (Seite 232), **difftime()** (Seite 187)

cos()

```
#include <math.h>

double cos( double phi );
```

liefert den cos(`phi`) (`phi` ist im Bogenmaß anzugeben).

Ist `phi` betragsmäßig zu groß, dann kann das Ergebnis ungenau sein, ohne daß `errno` gesetzt wird.

→**acos()** (Seite 177), **sin()** (Seite 222), **tan()** (Seite 232), **cosh()** (Seite 186)

cosh()

```
#include <math.h>

double cosh( double phi );
```

liefert den cosh(`phi`).

Ist `phi` betragsmäßig zu groß, dann wird der Wert `HUGE_VAL` zurückgegeben und `errno` auf `ERANGE` gesetzt.

→**cos()** (Seite 186), **sinh()** (Seite 223), **tanh()** (Seite 232)

ctime()

```
#include <time.h>

char *ctime( const time_t *zeit );
```

liefert einen Zeiger auf einen 26 Byte langen String (inkl. `'\0'`), in dem die Kalenderzeit `*zeit` als Text beschrieben ist. Der String wird beim nächsten Aufruf von `ctime()` oder `asctime()` überschrieben.

`ctime( zeit )` ist identisch mit `asctime( localtime( zeit ) )`; siehe dort.

→**asctime()** (Seite 177), **localtime()** (Seite 204), **time()** (Seite 232)

difftime()

```
#include <time.h>

double difftime( time_t zeit1, time_t zeit2 );
```

liefert die Differenz zwischen zwei Kalenderzeiten als `double` in der Einheit Sekunden. Wenn `zeit2` vor `zeit1` liegt, dann ist das Ergebnis negativ.

→**clock()** (Seite 185), **time()** (Seite 232)

div()

```
#include <stdlib.h>

div_t div( int dividend, int divisor );
```

führt die Division `dividend/divisor` aus und liefert eine Struktur mit zwei `int`-Elementen `quot` und `rem`. Das Element `quot` enthält das Ergebnis der ganzzahligen Division; `rem` enthält den Divisionsrest.

`divisor` darf nicht 0 sein.

→**ldiv()** (Seite 204), **fmod()** (Seite 192)

exit()

```
#include <stdlib.h>

void exit( int rueckgabe );
```

beendet das Programm, nachdem alle mit `atexit()` angemeldeten Funktionen ausgeführt wurden, alle offenen Dateien ordnungsgemäß geschlossen und alle temporären Dateien (siehe `tmpfile()`) gelöscht wurden.

An die Programmumgebung wird dann der Wert `rueckgabe` gemeldet. Ein Rückgabewert von 0 signalisiert allgemein, daß das Programm erfolgreich lief; andere Werte zeigen Fehler an. Dafür können auch die Werte `EXIT_SUCCESS` und `EXIT_FAILURE` aus stdlib.h verwendet werden.

→**abort()** (Seite 176), **atexit()** (Seite 179)

exp()

```
#include <math.h>

double exp( double wert );
```

liefert den Wert der Exponentialfunktion ($e^{\texttt{wert}}$). Wenn `wert` zu groß ist, dann wird `HUGE_VAL` geliefert und `errno` auf `ERANGE` gesetzt.

→**log()** (Seite 204)

fabs()

```
#include <math.h>

double fabs( double wert );
```

liefert den Absolutbetrag von `wert` zurück.

→**abs()** (Seite 176), **labs()** (Seite 203)

fclose()

```
#include <stdio.h>

int fclose( FILE *datei );
```

schließt die übergebene Datei, nachdem die Puffer freigegeben wurden. Wurde die Datei mit `tmpfile()` erzeugt, dann wird sie gelöscht.

Wenn alles klappt, gibt die Funktion 0 zurück; ansonsten liefert sie `EOF` und setzt `errno`, um den Fehler näher zu beschreiben.

Beispiele stehen ab der Seite 111 und auf der Seite 189.

→**fopen()** (Seite 192), **freopen()** (Seite 195), **tmpfile()** (Seite 234), **fflush()** (Seite 189)

feof()

```
#include <stdio.h>

int feof( FILE *datei );
```

testet den Indikator für das Dateiende der übergebenen Datei. Dieser Indikator kann mit `rewind()`, `fsetpos()`, `fseek()` oder `clearerr()` zurückgesetzt werden.

`feof()` liefert 0, wenn der Indikator für das Dateiende nicht gesetzt ist.

`feof()` kann als Makro definiert sein.

→**clearerr()** (Seite 185), **rewind()** (Seite 217), **fseek()** (Seite 197), **fsetpos()** (Seite 198)

ferror()

```
#include <stdio.h>

int ferror( FILE *datei );
```

Testet den Fehlerindikator der übergebenen Datei und liefert *wahr*, wenn selbiger gesetzt ist.

Der Indikator kann mit `rewind()` oder `clearerr()` gelöscht werden.

→**clearerr()** (Seite 185), **feof()** (Seite 188), **rewind()** (Seite 217)

fflush()

```
#include <stdio.h>

int fflush( FILE *datei );
```

schreibt eventuell noch im Speicher schwebende Puffer der übergebenen Datei aus. Dies wird bei normalem Programmende (`return` in `main()` oder `exit()`) oder beim Schließen der Datei oder beim Aufruf von `rewind()`, `fseek()` und `fsetpos()` automatisch ausgeführt.

Rückgabewert ist 0, wenn alles klappt. Ansonsten wird `EOF` zurückgegeben und `errno` entsprechend gesetzt.

→**rewind()** (Seite 217), **fseek()** (Seite 197), **fsetpos()** (Seite 198), **fclose()** (Seite 188), **exit()** (Seite 187)

fgetc()

```
#include <stdio.h>

int fgetc( FILE *datei );
```

liest das nächste Zeichen aus `datei` und liefert es zurück, wenn kein Fehler und kein Dateiende auftraten. Ansonsten wird `EOF` geliefert und `errno` sowie der entsprechende Indikator der Datei für Fehler oder Dateiende gesetzt.

Dieses Beispiel kopiert den Inhalt der Datei ein.dat zeichenweise in die Datei aus.dat:

```
/* fufgetc.c 30. 7.95 kw
 */

#include <stdio.h>

int main()
{
  int     zeichen;
  FILE
    *ein,
    *aus;

  /* Dateien oeffnen
```

```
   */
  ein = fopen( "ein.dat", "rb" );
  aus = fopen( "aus.dat", "wb" );

  /* bis Dateiende lesen und ...
   */
  while( (zeichen=fgetc( ein ))!=EOF )
  {
    /* ... schreiben
     */
    fputc( zeichen, aus );
  }

  /* Dateien wieder schliessen
   */
  fclose( ein );
  fclose( aus );
}
```

→**feof()** (Seite 188), **ferror()** (Seite 189), **clearerr()** (Seite 185), **fputc()** (Seite 194), **getc()** (Seite 199), **getchar()** (Seite 199)

fgetpos()

```
#include <stdio.h>

int fgetpos( FILE *datei, fpos_t *position_zeiger );
```

speichert die aktuelle Position von `datei` an der Speicherstelle `*position_zeiger`. Dieser Wert kann zu einem späteren Zeitpunkt an `fsetpos()` übergeben werden, um die Datei wieder an dieselbe Stelle zu positionieren.

Zurückgegeben wird 0, wenn alles gut ging. Ansonsten wird ein Wert ungleich 0 geliefert und `errno` wird gesetzt.

→**fsetpos()** (Seite 198), **ftell()** (Seite 198), **fseek()** (Seite 197)

fgets()

```
#include <stdio.h>

char *fgets( char *charfeld, int anzahl, FILE *datei );
```

versucht, aus `datei` einen String zu lesen.

Wenn das Dateiende gesetzt ist, bevor das erste Zeichen gelesen werden konnte, wird `NULL` geliefert und `charfeld` ist unverändert.

Wenn ein Lesefehler auftritt, wird ebenfalls `NULL` geliefert, aber charfeld kann verändert worden sein.

Ansonsten liefert `fgets()` einen Zeiger auf `charfeld[0]`, also den Wert von `charfeld`. Die Funktion hat dann Zeichen gelesen, bis entweder ein Zeichen mit dem Wert `'\n'` (Zeilenvorschubzeichen) oder `(anzahl-1)` Zeichen eingelesen wurden, oder bis das Dateiende erreicht ist. Die eingelesenen Zeichen werden mit einer `'\0'` beendet.

charfeld muß auf einen Speicherbereich zeigen, der für (anzahl) Zeichen Platz bietet.

→**fgetc()** (Seite 189), **fputs()** (Seite 194), **gets()** (Seite 199)

floor()

```
#include <math.h>

double floor( double wert );
```

liefert die nächstkleinere oder gleiche ganze Zahl zu wert als double.

Beispiel:

```
/* fufloor.c 30. 7.95 kw
 */

#include <math.h>

int main()
{
  double  d;

  for( d=-2.0; d<2.0; d+=0.2 )
    printf( "floor(%f) = %f\n", d, floor(d) );
}
```

Dieses Programm liefert die Ausgabe:

```
floor(-2.000000) = -2.000000
floor(-1.800000) = -2.000000
floor(-1.600000) = -2.000000
floor(-1.400000) = -2.000000
floor(-1.200000) = -2.000000
floor(-1.000000) = -1.000000
floor(-0.800000) = -1.000000
floor(-0.600000) = -1.000000
floor(-0.400000) = -1.000000
floor(-0.200000) = -1.000000
floor(0.000000) = 0.000000
floor(0.200000) = 0.000000
floor(0.400000) = 0.000000
floor(0.600000) = 0.000000
floor(0.800000) = 0.000000
floor(1.000000) = 1.000000
floor(1.200000) = 1.000000
floor(1.400000) = 1.000000
floor(1.600000) = 1.000000
floor(1.800000) = 1.000000
```

Hier treten beim Hochzählen von d Rundungsfehler auf; dadurch wird beispielsweise der Wert -1.0 nicht genau getroffen, sondern leicht unterschritten.

→**ceil()** (Seite 184)

fmod()

```
#include <math.h>

double fmod( double zaehler, double nenner );
```

liefert den Rest der Gleitkommadivision `zaehler/nenner` so, daß `zaehler` gleich dem Wert `fmod( zaehler, nenner ) + i*nenner` für irgendein `i` ist. Ist die Division nicht möglich (etwa weil `nenner` den Wert 0.0 hat), dann ist das Ergebnis undefiniert.

→**ceil()** (Seite 184), **floor()** (Seite 191), **modf()** (Seite 208)

fopen()

```
#include <stdio.h>

FILE *fopen( const char *dateiname, const char *modus );
```

öffnet eine Datei (sequentieller oder direkter Zugriff; Textdatei oder Binärdatei, Lesen oder Schreiben oder beides).

`modus` ist dabei einer der folgenden Strings:

"r"	"w"	"a"
"rb"	"wb"	"ab"
"r+"	"w+"	"a+"
"r+b"	"w+b"	"a+b"
"rb+"	"wb+"	"ab+"

Mit dem Zeichen `'r'` in `modus` öffnet man eine Datei zum Lesen. Mit `'w'` analog zum Schreiben; `'a'` zum Schreiben ab dem bisherigen Dateiende.

`"r+"`, `"w+"` und `"a+"` öffnen die Datei zum Ändern (Lesen und Schreiben), wobei `"a+"` die Datei so öffnet, daß sie anfangs auf dem bisherigen Dateiende positioniert ist. Bei `"r+"` (mit oder ohne `b`) muß die Datei bereits existieren, bei `"w+"` wird sie gegebenenfalls neu angelegt.

Ist in `modus` kein `b` enthalten, dann wird eine Textdatei erzeugt. Ist dagegen ein `b` enthalten, dann wird eine Binärdatei geöffnet.

Mit Textdateien sind dabei Dateien gemeint, die eine Zeilenstruktur aufweisen. In *C* werden die Zeilen solcher Dateien mit einem *line feed*-Zeichen, also einem Zeilenvorschub, abgeschlossen.

Text- und Binärdateien unterscheiden sich auf manchen Rechnern darin, daß Binärdateien tatsächlich nur die Zeichen enthalten, die hineingeschrieben wurden und man beim Lesen nur die Zeichen erhält, die wirklich in der Datei stehen. Textdateien enthalten dagegen als Zeilentrenner auf manchen Rechnersystemen die Zeichenfolge `'\r'`, `'\n'`; also *Return* und *Linefeed*. Auf anderen Rechnern (insbesondere Unix) ist als Zeilentrenner nur `'\n'` (*Linefeed*) üblich. Um jetzt diese Rechnerabhängigkeit vom *C*-Quelltext fernhalten zu können, öffnet man Textdateien ohne ein `b` in `modus`. Als Zeilentrenner (z.B. in `fprintf(...)`) verwendet man auf jedem System innerhalb des Quelltextes nur das Zeichen `'\n'`. Dann wird —je nach Rechner— in den verwendeten Bibliotheksfunktionen jedes geschriebene Zeichen gegebenenfalls daraufhin untersucht, ob es ein `'\n'` ist. Wenn auf dem Rechner nur `'\n'` als Zeilentrenner üblich ist, dann werden alle Zeichen unverändert weitergegeben und geschrieben. Im anderen Fall werden alle `'\n'` in die Zeichenfolge `'\r'`, `'\n'` verwandelt. Umgekehrt werden beim Lesen aus Textdateien eventuell alle `'\r'`, `'\n'` in das eine Zeichen `'\n'` verwandelt, so daß das eigentliche Programm die Zeichenfolge *Return*, *Linefeed* gar nicht sieht.

Binärdateien dagegen sind von dieser Unterscheidung ausgenommen, hier wird unverändert übertragen.

Wird eine Datei nur zum Lesen geöffnet, dann muß sie bereits existieren.

Wird die Datei zum Schreiben oder Ändern geöffnet und existiert noch nicht, dann wird sie (leer) erzeugt.

Wird die Datei zum Schreiben geöffnet und existiert schon, dann wird der alte Inhalt gelöscht.

Im Änderungsmodus geöffnete Dateien dürfen wechselweise beschrieben und gelesen werden. Der Wechsel zwischen Lesen und Schreiben (oder umgekehrt) muß aber unbedingt von einem Aufruf der Funktionen `rewind()`, `fseek()`, `fsetpos()` oder `fflush()` für die betreffende Datei begleitet sein, da sonst die Pufferung der Datei schief geht.

Wenn die Datei wie gewünscht geöffnet werden konnte, dann gibt `fopen()` einen Zeiger auf die Dateibeschreibung zurück. Dieser Zeiger darf nicht verändert werden und wird für alle nachfolgenden Operationen mit dieser Datei bis zum `fclose()` oder dem Programmende benötigt.

Kann die Datei nicht geöffnet werden, dann gibt `fopen()` den Wert `NULL` zurück und setzt `errno`.

Beispiele stehen unter anderem auf der Seite 111 und auf der Seite 189.

→**fclose()** (Seite 188), **freopen()** (Seite 195), **fread()** (Seite 194),
→**fwrite()** (Seite 198), **fprintf()** (Seite 193), **fscanf()** (Seite 196),
→**fputc()** (Seite 194), **fputs()** (Seite 194), **fgetc()** (Seite 189),
→**fgets()** (Seite 190), **fgetpos()** (Seite 190), **fsetpos()** (Seite 198),
→**ftell()** (Seite 198), **fseek()** (Seite 197)

fprintf()

```
#include <stdio.h>

int fprintf( FILE *datei, const char *format, ... );
```

schreibt formatierte Ausgabe in die Datei `datei`.

Abgesehen von dem zusätzlichen Parameter `datei`, der die zu beschreibende Datei angibt, ist `fprintf()` identisch mit `printf()`, siehe also dort.

Zurückgegeben wird die Anzahl der geschriebenen Bytes (nicht eingerechnet sind in diese Anzahl die zusätzlichen Zeichen, die durch eine eventuelle Umwandlung für Textdateien auftreten; siehe dazu die Beschreibung in `fopen()`).

→**fopen()** (Seite 192), **printf()** (Seite 210), **sprintf()** (Seite 223),
→**vfprintf()** (Seite 238)

fputc()

```
#include <stdio.h>

int fputc( int zeichen, FILE *datei );
```

verwandelt `zeichen` in den Typ `char` und schreibt das Ergebnis nach `datei`.

Zurückgegeben wird das geschriebene Zeichen oder `EOF`, wenn das Schreiben nicht möglich war. Dann werden `errno` und der Fehlerindikator in der Struktur `*datei` gesetzt.

→**fopen()** (Seite 192), **putc()** (Seite 212), **putchar()** (Seite 212),
→**fputs()** (Seite 194), **fgetc()** (Seite 189), **getc()** (Seite 199)

fputs()

```
#include <stdio.h>

int fputs( const char *string, FILE *datei );
```

versucht, den String, der ab `*string` im Speicher steht, nach datei zu schreiben bis die abschließende Null erreicht ist. Diese wird nicht übertragen.

Wenn das Schreiben gelingt, dann wird der Wert 0 zurückgegeben. Beim Auftreten eines Fehlers wird ein Wert ungleich 0 geliefert und der Fehlerindikator in `*datei` und `errno` werden gesetzt.

Im Gegensatz zu `puts()` wird an den auszugebenden String nicht automatisch ein Zeilenvorschub angehängt; dies könnte mit
`fprintf( datei, "%s\n", string )`
anstatt
`fputs( string, datei )`
erreicht werden.

→**puts()** (Seite 213), **fgets()** (Seite 190), **fputs()** (Seite 194),
→**fprintf()** (Seite 193)

fread()

```
#include <stdio.h>

size_t fread( void *wohin,
              size_t elementgroesse,
              size_t elemente,
              FILE *datei            );
```

versucht aus der Datei `datei` in den Speicher ab `wohin` zu lesen, bis entweder maximal `(elementgroesse * elemente)` Zeichen in den Speicher übertragen wurden oder das Dateiende erreicht ist oder ein Fehler auftritt. Eine eventuelle Anpassung der Daten für Textdateien (siehe `fopen()`) wird dabei durchgeführt.

Zurückgegeben wird die Anzahl der eingelesenen Elemente. Das ist maximal der Wert von `elemente`, wenn alles gut ging, sonst weniger. Dann wird der Indikator für Dateiende von `datei` oder der Fehlerindikator gesetzt; in letzterem Fall auch `errno`.

→**fopen()** (Seite 192), **fwrite()** (Seite 198), **feof()** (Seite 188),
→**ferror()** (Seite 189)

free()

```
#include <stdlib.h>

void free( void *p )
```

gibt den Speicher ab `p` frei, der zuvor mit `malloc()` oder `calloc()` allokiert worden sein muß.

Ein Beispiel steht auf Seite 49.

→**malloc()** (Seite 206), **calloc()** (Seite 184), **realloc()** (Seite 216)

freopen()

```
#include <stdio.h>

FILE *freopen( const char *dateiname,
               const char *modus,
               FILE *datei            );
```

verknüpft eine Datei `dateiname` mit einer schon offenen Datei oder einem Gerät `datei`.

`datei` wird dazu eventuell geschlossen und als `dateiname` neu geöffnet.

Zu `modus` siehe `fopen()`.

Diese Funktion wird manchmal dazu verwendet, einen der vordefinierten Gerätekanäle `stdin`, `stdout`, `stderr` auf eine Datei umzubiegen.

Rückgabe ist der übergebene Zeiger `datei`, wenn die Umlenkung klappte, oder sonst der Wert `NULL`.

Zum Beispiel kann man mit
`freopen( dateiname, "w", stdout )`
alle nachfolgenden Ausgaben des Programms, die über `stdout` gehen, in eine Datei umbiegen:

```
/* fufreopn.c 30. 7.95 kw
 */

#include <stdio.h>

main()
{
  freopen( "test.dat", "w", stdout );
  printf( "dieser Text landet in der Datei test.dat!\n" );
}
```

Je nach Rechnersystem kann man diese Verknüpfung wieder aufheben, wenn man die Standardausgabe unter ihrem Gerätenamen wieder öffnet. Die folgende Sequenz leitet unter DOS die Ausgabe in eine Datei um und dann wieder zurück auf den Bildschirm:

```
        freopen( "test.dat", "w", stdout );
        printf( "dieser Text landet in der Datei test.dat!\n" );
        freopen( "CON", "w", stdout );
        printf( "und das wieder auf dem Bildschirm!\n" );
```

→**fopen()** (Seite 192), **fclose()** (Seite 188)

frexp()

```
#include <math.h>

double frexp( double wert, int *exp_zeiger );
```

zerlegt den Parameter `wert` in eine normalisierte Mantisse im Bereich 0.5...1.0 und einen Binärexponenten. Die Mantisse wird zurückgegeben. Der Exponent wird an die Stelle im Speicher geschrieben, auf die `exp_zeiger` zeigt.

Die Zerlegung geschieht so, daß die Mantisse multipliziert mit 2^{*exp_zeiger} den übergebenen Parameter `wert` ergibt.

Wenn `wert` gleich 0.0 ist, dann sind beide Teile der Zerlegung 0.

Beispiel:

```
/* fufrexp.c 30. 7.95 kw
 */

#include <math.h>

main()
{
  int    expo;
  double frac;

  frac = frexp( 12.3456, &expo );
  printf( "frac = %20.10f; expo = %d\n", frac, expo );
  printf( "frac*(2^expo) = %20.10f\n", frac*pow(2.0,expo) );
}
```

erzeugt die Ausgabe

```
frac =         0.7716000000; expo = 4
frac*(2^expo) =        12.3456000000
```

→**ldexp()** (Seite 203), **modf()** (Seite 208)

fscanf()

```
#include <stdio.h>

int fscanf( FILE *datei, const char *format, ... );
```

liest aus einer Datei formatierte Eingabe und wandelt sie nach den Anweisungen in `format` um.

Die weiteren Parameter richten sich nach den Direktiven in `format` und sind Zeiger auf Speicherplätze, an welche die umgewandelten Werte geschrieben werden können.

Bis auf den zusätzlichen Parameter `datei`, der angibt, aus welcher Datei gelesen werden soll, ist `fscanf()` identisch mit `scanf()`, siehe also dort.

Zurückgegeben wird die Anzahl der gelesenen und umgewandelten Werte oder `EOF`, wenn das Dateiende vor der ersten Übertragung erreicht wurde.

→**scanf()** (Seite 218), **sscanf()** (Seite 224), **fprintf()** (Seite 193)

fseek()

```
#include <stdio.h>

int fseek( FILE *datei, long int wieweit, int abwo );
```

positioniert die Datei `datei` an eine neue Position.

Je nach dem Wert von `abwo` zählt die Positionsangabe `wieweit`

- ab dem Dateianfang, wenn `abwo` gleich `SEEK_SET` ist,
- ab der aktuellen Position, wenn `abwo` gleich `SEEK_CUR` ist, oder
- ab dem Dateiende, wenn `abwo` gleich `SEEK_END` ist.

`SEEK_SET`, `SEEK_CUR` und `SEEK_END` sind in stdio.h definiert.

Bei der Positionierung wird der Indikator für Dateiende gelöscht und die Wirkung eines eventuellen vorherigen Aufrufs der Funktion `ungetc()` wird aufgehoben. Durch den Aufruf von `fseek()` darf bei Dateien im Änderungsmodus zwischen Lese- und Schreibzugriff gewechselt werden oder umgekehrt.

Achtung! Bei Dateien im Textmodus ist `fseek()` nur sinnvoll, wenn entweder

- `wieweit` von einem vorherigen Aufruf von `ftell()` stammt und `abwo` den Wert `SEEK_SET` hat, oder
- `wieweit` den Wert 0 hat.

Bei allen anderen Kombinationen können in Textmodus Fehler auftreten, weil die Positionen, die das Programm sieht, nicht identisch mit physikalischen Positionen in der Datei sein müssen (siehe `fopen()`).

Im erfolgreichen Fall gibt `fseek()` den Wert 0 zurück. Ansonsten wird ein Wert ungleich 0 geliefert und `errno` gesetzt.

→**ftell()** (Seite 198), **fopen()** (Seite 192), **fgetpos()** (Seite 190),
→**fsetpos()** (Seite 198)

fsetpos()

```
#include <stdio.h>

int fsetpos( FILE *datei, fpos_t *position );
```

setzt die Datei `datei` auf die Position `*position`, die von einem vorherigen Aufruf von `fgetpos()` stammen muß.

Der Dateiende-Indikator der Datei wird gelöscht; ein eventueller vorheriger Aufruf der Funktion `ungetc()` wird unwirksam. Weiterhin erlaubt `fsetpos()` bei Dateien im Änderungsmodus den Wechsel zwischen Ein- und Ausgabe.

Geht alles gut, dann liefert `fsetpos()` den Wert 0; Ansonsten einen anderen Wert, weil die Position ungültig war. Dann wird `errno` auf den Wert `EINVAL` gesetzt.

→**fgetpos()** (Seite 190), **fseek()** (Seite 197), **ftell()** (Seite 198)

ftell()

```
#include <stdio.h>

long int ftell( FILE *datei );
```

liefert die momentane Position der Datei `datei` zurück.

Dieser Wert kann für spätere Aufrufe von `fseek( datei, ..., SEEK_SET )` verwendet werden (sowohl für Text- als auch Binärdateien).

Für Binärdateien ist der Rückgabewert die Dateiposition.

Im Fehlerfall wird `-1L` zurückgegeben und `errno` wird gesetzt.

→**fseek()** (Seite 197), **fopen()** (Seite 192), **fgetpos()** (Seite 190),
→**fsetpos()** (Seite 198)

fwrite()

```
#include <stdio.h>

size_t fwrite( const void *woher,
               size_t elementgroesse,
               size_t elemente,
               FILE *datei            );
```

schreibt ab der Speicherstelle `woher` maximal (`elementgroesse * elemente`) Bytes in die Datei `datei`.

Bei Dateien im Textmodus wird die übliche Konvertierung (siehe `fopen()`) durchgeführt.

Zurückgegeben wird die Anzahl der tatsächlich geschriebenen Elemente. Im Fehlerfall ist dies weniger als der Wert von `elemente`; dann werden der Fehlerindikator der Datei und `errno` gesetzt.

→**fopen()** (Seite 192), **fread()** (Seite 194), **fprintf()** (Seite 193),
→**fputs()** (Seite 194)

getc()

```
#include <stdio.h>

int getc( FILE *datei );
```

liest aus `datei` das nächste Zeichen.

Geliefert wird das gelesene Zeichen oder `EOF`, wenn das Dateiende erreicht wurde, oder wenn ein Fehler auftrat. In diesen beiden Fällen wird der Indikator für Dateifehler bzw. für Dateiende in `datei` gesetzt.

Achtung! Je nach System kann `getc()` als Makro definiert sein. Sicherer ist die Verwendung der gleichwertigen Funktion `fgetc()`.

→**fgetc()** (Seite 189), **fopen()** (Seite 192), **getchar()** (Seite 199),
→**ungetc()** (Seite 235)

getchar()

```
#include <stdio.h>

int getchar( void );
```

liest ein Zeichen von der Standardeingabe `stdin` und ist identisch mit `getc( stdin )`. Siehe dort.

→**fgetc()** (Seite 189), **getc()** (Seite 199), **ungetc()** (Seite 235)

getenv()

```
#include <stdlib.h>

char *getenv( const char *name );
```

sucht im sogenannten *environment* (siehe auch **Environmentvariablen** (Seite 107)) des Programms nach einer Variable mit der Bezeichnung `*name`. Wird eine solche gefunden, dann liefert `getenv()` einen Zeiger auf ihren Wert; ansonsten `NULL`.

Der Variablenname ab (`*name`) sollte in Großbuchstaben angegeben werden.

→**system()** (Seite 232)

gets()

```
#include <stdio.h>

char *gets( char *s );
```

liest einen String von der Standardeingabe in den Speicher ab der Stelle, auf die `s` zeigt, bis das Dateiende von `stdin` auftritt, oder ein Fehler, oder ein Zeilenvorschub gelesen wird. Der Zeilenvorschub wird nicht mitgespeichert. Anschließend wird ein Zeichen mit dem Wert `\0` angehängt.

Zurückgegeben wird der Zeiger `s`, wenn das Lesen klappte.

Ist das Dateiende schon vor dem Lesen erreicht oder tritt ein Dateifehler auf, dann wird Null zurückgegeben und bei einem Fehler `errno` gesetzt.

Achtung! Im Gegensatz zu `fgets()` wird keine Feldüberschreitung geprüft!

→**puts()** (Seite 213), **fgets()** (Seite 190), **getchar()** (Seite 199),
→**fgetc()** (Seite 189)

gmtime()

```
#include <time.h>

struct tm *gmtime( const time_t *zeit_zeiger );
```

verwandelt die Kalenderzeit in `*zeit_zeiger` in eine Struktur als *Greenwich mean time* (GMT). Die gegebene Zeit sollte als Ortszeit vorliegen (etwa aus `time()`). Dazu wird die Zeitdifferenz zwischen GMT und der Ortszeit aus der environment-Variablen `TZ` bestimmt. Dieser String sollte aus drei Buchstaben bestehen, welche die Standardzeitzone beschreiben, weiterhin aus einer vorzeichenbehafteten ganzen Zahl, die den Abstand angibt, um den die Ortszeit der GMT vorausgeht. Falls Sommerzeit ist, sollten anschließend drei Buchstaben folgen, die den Namen der Sommerzeitzone angeben.

Zurückgegeben wird ein Zeiger auf eine statische Struktur, in der die errechnete Zeit abgelegt wird. Diese Struktur wird bei jedem Aufruf von
`gmtime()`, `asctime()`, `ctime()` und `localtime()`
überschrieben.

Wenn die environment-Variable `TZ` nicht existiert, dann wird `NULL` zurückgegeben.

→**asctime()** (Seite 177), **ctime()** (Seite 187), **localtime()** (Seite 204),
→**time()** (Seite 232), **mktime()** (Seite 208)

isalnum()

```
#include <ctype.h>

int isalnum( int c );
```

liefert ungleich 0, wenn `c` ein alphanumerisches Zeichen ist (`0`–`9`, `a`–`z`, `A`–`Z`).

Deutsche Umlaute werden nicht als Buchstaben behandelt, sondern als Sonderzeichen.

→**isalpha()** (Seite 200), **iscntrl()** (Seite 201), **isdigit()** (Seite 201),
→**isgraph()** (Seite 201), **islower()** (Seite 202), **isprint()** (Seite 202),
→**ispunct()** (Seite 202), **isspace()** (Seite 202), **isupper()** (Seite 203),
→**isxdigit()** (Seite 203)

isalpha()

```
#include <ctype.h>

int isalpha( int c );
```

liefert ungleich 0, wenn `c` ein Buchstabe ist (`a`–`z`, `A`–`Z`).

Deutsche Umlaute werden nicht als Buchstaben behandelt, sondern als Sonderzeichen.

→**isalnum()** (Seite 200), **iscntrl()** (Seite 201), **isdigit()** (Seite 201),
→**isgraph()** (Seite 201), **islower()** (Seite 202), **isprint()** (Seite 202),
→**ispunct()** (Seite 202), **isspace()** (Seite 202), **isupper()** (Seite 203),
→**isxdigit()** (Seite 203)

iscntrl()

```
#include <ctype.h>

int iscntrl( int c );
```

liefert ungleich 0, wenn `c` ein Steuerzeichen ist (Im ASCII-Zeichensatz sind das die Zeichen 0 bis 31 dezimal).

→isalnum, isalpha, isdigit,
→isgraph, islower, isprint, ispunct,
→isspace, isupper, isxdigit

isdigit()

```
#include <ctype.h>

int isdigit( int c );
```

liefert ungleich 0, wenn `c` eine dezimale Ziffer ist (`0`–`9`).

→**isalnum()** (Seite 200), **isalpha()** (Seite 200), **iscntrl()** (Seite 201),
→**isgraph()** (Seite 201), **islower()** (Seite 202), **isprint()** (Seite 202),
→**ispunct()** (Seite 202), **isspace()** (Seite 202), **isupper()** (Seite 203),
→**isxdigit()** (Seite 203)

isgraph()

```
#include <ctype.h>

int isgraph( int c );
```

liefert ungleich 0, wenn `c` ein graphisches (nicht druckbares) Zeichen ist.

→**isalnum()** (Seite 200), **isalpha()** (Seite 200), **iscntrl()** (Seite 201),
→**isdigit()** (Seite 201), **islower()** (Seite 202), **isprint()** (Seite 202),
→**ispunct()** (Seite 202), **isspace()** (Seite 202), **isupper()** (Seite 203),
→**isxdigit()** (Seite 203)

islower()

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

```
int islower( int c );
```

liefert ungleich 0, wenn `c` ein Kleinbuchstabe ist (`a`–`z`).

Deutsche Umlaute werden nicht als Buchstaben behandelt, sondern als Sonderzeichen.

→**isalnum()** (Seite 200), **isalpha()** (Seite 200), **iscntrl()** (Seite 201),
→**isdigit()** (Seite 201), **isgraph()** (Seite 201), **isprint()** (Seite 202),
→**ispunct()** (Seite 202), **isspace()** (Seite 202), **isupper()** (Seite 203),
→**isxdigit()** (Seite 203)

isprint()

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

```
int isprint( int c );
```

liefert ungleich 0, wenn `c` ein druckbares Zeichen oder ein Leerzeichen ist.

→**isalnum()** (Seite 200), **isalpha()** (Seite 200), **iscntrl()** (Seite 201),
→**isdigit()** (Seite 201), **isgraph()** (Seite 201), **islower()** (Seite 202),
→**ispunct()** (Seite 202), **isspace()** (Seite 202), **isupper()** (Seite 203),
→**isxdigit()** (Seite 203)

ispunct()

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

```
int ispunct( int c );
```

liefert ungleich 0, wenn `c` ein Interpunktionszeichen ist; dazu zählen alle druckbaren Zeichen (außer dem Leerzeichen), die nicht alphanumerisch sind (also auch deutsche Umlaute).

→**isalnum()** (Seite 200), **isalpha()** (Seite 200), **iscntrl()** (Seite 201),
→**isdigit()** (Seite 201), **isgraph()** (Seite 201), **islower()** (Seite 202),
→**isprint()** (Seite 202), **isspace()** (Seite 202), **isupper()** (Seite 203),
→**isxdigit()** (Seite 203)

isspace()

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

```
int isspace( int c );
```

liefert ungleich 0, wenn `c` ein sogenanntes *white-space*-Zeichen ist (also `\t`, `␣` (*Leerzeichen*), `\f`, `\n`, `\r`, `\v`);

→**isalnum()** (Seite 200), **isalpha()** (Seite 200), **iscntrl()** (Seite 201),
→**isdigit()** (Seite 201), **isgraph()** (Seite 201), **islower()** (Seite 202),
→**isprint()** (Seite 202), **ispunct()** (Seite 202), **isupper()** (Seite 203),
→**isxdigit()** (Seite 203)

isupper()

```
#include <ctype.h>

int isupper( int c );
```

liefert ungleich 0, wenn `c` ein Großbuchstabe ist (`A`–`Z`).

Deutsche Umlaute werden nicht als Buchstaben behandelt, sondern als Sonderzeichen.

→**isalnum()** (Seite 200), **isalpha()** (Seite 200), **iscntrl()** (Seite 201),
→**isdigit()** (Seite 201), **isgraph()** (Seite 201), **islower()** (Seite 202),
→**isprint()** (Seite 202), **ispunct()** (Seite 202), **isspace()** (Seite 202),
→**isxdigit()** (Seite 203)

isxdigit()

```
#include <ctype.h>

int isxdigit( int c );
```

liefert ungleich 0, wenn `c` eine hexadezimale Ziffer ist (also `0`–`9`, `a`–`f`, `A`–`F`).

→**isalnum()** (Seite 200), **isalpha()** (Seite 200), **iscntrl()** (Seite 201),
→**isdigit()** (Seite 201), **isgraph()** (Seite 201), **islower()** (Seite 202),
→**isprint()** (Seite 202), **ispunct()** (Seite 202), **isspace()** (Seite 202),
→**isupper()** (Seite 203)

labs()

```
#include <stdlib.h>

long int labs( long int wert );
```

gibt den Absolutbetrag von `wert` zurück.

→**abs()** (Seite 176), **fabs()** (Seite 188)

ldexp()

```
#include <math.h>

double ldexp( double wert, int expo );
```

multipliziert `wert` mit der Zweierpotenz $2^{\texttt{expo}}$. Das Ergebnis wird zurückgegeben.

Wenn das Ergebnis so groß ist, daß es nicht als `double` dargestellt werden kann, dann wird `HUGE_VAL` geliefert und `errno` auf den Wert `ERANGE` gesetzt.

→**frexp()** (Seite 196)

ldiv()

```
#include <stdlib.h>

ldiv_t ldiv( long int dividend, long int divisor );
```

führt die Division `dividend/divisor` aus und liefert eine Struktur mit den zwei `long int`-Elementen `quot` und `rem`. Das Element `quot` enthält das Ergebnis der ganzzahligen Division; `rem` enthält den Divisionsrest.

`divisor` darf nicht 0 sein.

→**div()** (Seite 187), **fmod()** (Seite 192)

localtime()

```
#include <time.h>

struct tm *localtime( const time_t *zeit_zeiger );
```

verwandelt die gepackte (lokale) Kalenderzeit `*zeit_zeiger` in die aufgebrochene Form `struct tm` und liefert einen Zeiger auf diese statische Struktur. Diese Struktur wird bei jedem Aufruf von
`gmtime()`, `asctime()`, `ctime()` und `localtime()`
überschrieben.

Die Kalenderzeit kann man mit `time()` erhalten.

Ein Beispiel dazu steht auf der Seite 232.

→**gmtime()** (Seite 200), **ctime()** (Seite 187), **asctime()** (Seite 177), →**mktime()** (Seite 208), **time()** (Seite 232)

log()

```
#include <math.h>

double log( double argument );
```

berechnet den natürlichen Logarithmus ln(`argument`) (also zur Basis e).

Wenn `argument==0.0` ist, dann tritt ein Resultatfehler auf, das heißt `errno` wird zu `ERANGE` gesetzt und `-HUGE_VAL` wird zurückgegeben.

Wenn `argument` kleiner als 0.0 ist, dann wird ein Argumentfehler auftreten und `errno` den Wert `EDOM` erhalten. Der Funktionswert ist dann 0.0.

→**log10()** (Seite 205)

log10()

```
#include <math.h>

double log10( double argument );
```

berechnet den dekadischen Logarithmus log(`argument`) (also zur Basis 10).

Wenn `argument==0.0` ist, dann tritt ein Resultatfehler auf, das heißt `errno` wird zu `ERANGE` gesetzt und `-HUGE_VAL` wird zurückgegeben.

Wenn `argument` kleiner als 0.0 ist, dann wird ein Argumentfehler auftreten und `errno` den Wert `EDOM` erhalten. Der Funktionswert ist dann 0.0.

→**log()** (Seite 204)

longjmp()

```
#include <setjmp.h>

void longjmp( jmp_buf umgebung, int return_wert );
```

springt zu dem Funktionsaufruf zurück, der in der Struktur `umgebung` von einem vorhergehenden Aufruf von `setjmp()` gespeichert ist.

Die Programmausführung wird nicht mit der Anweisung nach `longjmp()` weitergesetzt, sondern mit der Anweisung nach dem zu `umgebung` gehörenden `setjmp()`-Aufruf!

Aus dem genannten `setjmp()`-Aufruf heraus wird das Programm mit dem Rückgabewert `return_wert` fortgesetzt. Deshalb sollte `return_wert` nicht den Wert 0 haben, da sonst der Aufrufer nicht den Rücksprung von `longjmp()` erkennen kann. Wenn `return_wert` trotzdem auf 0 gesetzt ist, wird automatisch mit 1 aus dem `setjmp()`-Aufruf zurückgesprungen.

Achtung! Der zu einem Aufruf von `longjmp()` gehörige `setjmp()`-Aufruf muß:

1) zeitlich (bezüglich des Programmlaufs) *vor* `longjmp()` erfolgen, und
2) eine der beiden folgenden Bedingungen erfüllen:
   a) in derselben Funktion stehen wie `longjmp()` (dann wäre ein `goto` aber zweckmäßiger), oder
   b) in der Aufrufhierarchie in direkter Linie oberhalb von `longjmp()` stehen.

   Es ist also nicht möglich, in einer Funktion `setjmp()` aufzurufen, die Funktion zu verlassen und *dann* mit `longjmp()` wieder dorthin zurückzukehren! Dies liegt daran, daß mit dem Rücksprung aus der Funktion auch der sogenannte *Stack* ungültig wird, auf dem alle Funktionsparameter und lokalen Variablen abgelegt werden (vergleiche **9 Speichermodell eines Prozesses** auf Seite 73). Wird eine Funktion verlassen und kehrt man danach mit `longjmp()` wieder dorthin zurück, dann hat die Funktionen einerseits ihre (automatischen) Variablen und Parameter verloren, und hat andererseits keine berechtigte Chance, zu ihrem Aufrufer zurückkehren zu können.

Nochmal Achtung! Automatische Variablen (also lokale Variablen, die nicht als `static` vereinbart sind) werden bei einem `longjmp()`-Sprung nicht unbedingt restauriert (je nach System)! Will man solche Variablen restauriert haben, muß man sie als `volatile` (siehe Seite 52) deklarieren.

Beispiel:

```
/* fulongjm.c 30. 7.95 kw
 */

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

jmp_buf umgebung;

void f( void )
{
  puts( "Hier ist f()." );
  puts( "Wollen Sie f() mit longjmp() verlassen?" );
  if( toupper(getchar())=='J' )
  {
    longjmp( umgebung, 1 );
  }

  puts( "Na gut, dann nicht!" );
  return;
}

int main()
{
  int     wert;

  wert = setjmp( umgebung );

  /* Nach dem Aufruf von setjmp() hat wert den Inhalt 0.
   * Nach dem Ruecksprung durch longjmp() ist wert
   * dagegen ungleich 0.
   */
  if( wert==0 )
  {
    puts( "hier ist main(), gleich kommt f()!" );
    f();
    puts( "hier ist main(); "
          "f() wurde mit return verlassen!" );
  }
  else
  {
    puts( "hier ist main() nach longjmp()!" );
  }
  return 0;
}
```

→**setjmp()** (Seite 221)

malloc()

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

```
void *malloc( size_t wieviel );
```

versucht, einen Speicherblock mit `wieviel` Bytes zu reservieren.

Rückgabewert ist ein Zeiger auf den Anfang des reservierten Blocks, wenn alles gut ging, oder der `NULL`-Zeiger, wenn kein Block ausreichender Größe zu haben ist.

Der zugewiesene Speicher wird nicht gelöscht, sondern hat einen zufälligen Inhalt[39].

Ein Beispiel steht auf Seite 49.

→**free()** (Seite 195), **calloc()** (Seite 184), **realloc()** (Seite 216)

memchr()

```
#include <string.h>
```

```
void *memchr( const void *abwo, int zeichen, size_t wieviel );
```

durchsucht ab der Speicherstelle `abwo` maximal `wieviel` Zeichen nach `zeichen` und liefert einen Zeiger auf das erste Vorkommen oder `NULL`, wenn `zeichen` in dem Bereich nicht gefunden wird.

→**strchr()** (Seite 224)

memcmp()

```
#include <string.h>
```

```
int memcmp( const void *zeiger1, const void *zeiger2, size_t wieviele );
```

vergleicht den Speicher ab der Speicherstelle `zeiger1` mit dem Speicher ab `zeiger2`.

Nach maximal `wieviele` Bytes wird der Vergleich abgebrochen. Wenn bis dahin kein Unterschied festgestellt wurde, wird der Wert 0 zurückgegeben. Ansonsten liefert der Aufruf von `memcmp()` einen Wert größer, gleich oder kleiner als 0, je nachdem ob das erste unterschiedliche Zeichen im Bereich ab `zeiger1` größer, gleich oder kleiner als das entsprechende Zeichen im Bereich ab `zeiger2` ist.

→**strcmp()** (Seite 224), **strncmp()** (Seite 228)

memcpy()

```
#include <string.h>
```

```
void *memcpy( void *wohin, void *woher, size_t wieviele );
```

kopiert `wieviele` Bytes ab der Speicherstelle `woher` in den Bereich ab `wohin`. Die Bereiche sollten sich nicht überschneiden (für diesen Fall gibt es `memmove()`).

`memcpy()` liefert den Zeiger `wohin` zurück.

→**memmove()** (Seite 208), **strcpy()** (Seite 225), **strncpy()** (Seite 228)

[39] nicht so bei `calloc()`, siehe dort.

memmove()

```
#include <string.h>

void *memmove( void *wohin, void *woher, size_t wieviele );
```

kopiert einen Speicherbereich an eine andere Stelle im Speicher.

Dabei darf sich der Zielbereich (`wohin` bis `(wohin + wieviele)`) mit dem Quellbereich (`woher` bis `(woher + wieviele)`) überlappen.

Ist sichergestellt, daß sich die Bereiche nicht überlappen, dann ist die Funktion `memcpy()` gleichwertig, aber schneller.

Zurückgegeben wird ein Zeiger auf den Zielbereich, also `wohin`.

→**memcpy()** (Seite 207), **strcpy()** (Seite 225), **strncpy()** (Seite 228)

memset()

```
#include <string.h>

void *memset( void *abwo, int womit, size_t wieviele );
```

füllt ab der Stelle `abwo` im Arbeitsspeicher `wieviele` aufeinanderfolgende Bytes mit dem Zeichen `(unsigned char)womit`.

Zurückgegeben wird ein Zeiger auf das gefüllte Objekt, also `abwo`.

→**calloc()** (Seite 184)

mktime()

```
#include <time.h>

time_t mktime( struct tm *zeitzeiger );
```

konvertiert die Ortszeit aus der Struktur `*zeitzeiger` in das komprimierte Format der Funktion `time()`.

Dies ist die Umkehrung von `localtime()`.

→**localtime()** (Seite 204), **time()** (Seite 232), **gmtime()** (Seite 200)

modf()

```
#include <math.h>

double modf( double wert, double *ganzzahliger_teil );
```

zerlegt `wert` in einen ganzzahligen Anteil und einen gebrochenen Teil, der zurückgegeben wird. Der ganzzahlige Teil wird bei `ganzzahliger_teil` hinterlegt. Beide Teile haben das gleiche Vorzeichen wie `wert`.

→**floor()** (Seite 191), **ceil()** (Seite 184), **fmod()** (Seite 192), **frexp()** (Seite 196)

perror()

```
#include <stdio.h>

void perror( const char *text );
```

Wenn `text` ungleich `NULL` ist, dann wird der String ab `text` nach `stderr` geschrieben.

Anschließend wird eine Fehlermeldung ausgegeben, die dem aktuellen Wert von `errno` entspricht. Anschließend wird ein Zeilenvorschub ausgegeben.

→**strerror()** (Seite 225)

pow()

```
#include <math.h>

double pow( double basis, double exponent );
```

gibt die Potenz (`basis`)$^{\texttt{exponent}}$ zurück, wenn möglich.

Ist das Ergebnis zu groß, um als `double` dargestellt zu werden, dann wird `+HUGE_VAL` oder `-HUGE_VAL` geliefert und `errno` auf `ERANGE` gesetzt.

Wenn:

- `basis` negativ ist und `exponent` nicht ganzzahlig oder
- `basis` ist 0.0 und `exponent` ist kleiner oder gleich 0.0,

(Argumentfehler), dann wird 0.0 zurückgegeben und `errno` auf `EDOM` gesetzt.

Wenn man nicht unbedingt gebrochene Exponenten braucht, dann kann man Potenzen eventuell schneller berechnen. Siehe dazu Abschnitt **A.2 Potenzen mit ganzzahligen Exponenten** auf Seite 245.

→**log()** (Seite 204), **log10()** (Seite 205), **frexp()** (Seite 196)

printf()

```
#include <stdio.h>

int printf( char *format, ... )
```

gibt den String `format` aus. Sind darin Prozentzeichenfolgen `%d`, `%f`, `%s`, und so weiter enthalten, dann werden an deren Stelle weitere Werte ausgegeben, die in der gleichen Reihenfolge hinter dem Parameter `format` folgen müssen, wie die zugehörigen Prozentzeichenfolgen in `format`.

Der String, auf den der Parameter `format` zeigt, wird auf die Standardausgabe `stdout` ausgegeben. Dabei dürfen die üblichen Steuerzeichen wie `\t` als Tabulator, `\n` als Zeilenvorschub und `\a` als Piepston eingestreut sein.

Ein Prozentzeichen `%` im Formatstring leitet eine Veränderung der Ausgabe ein.

Nach dem Prozentzeichen folgt eine der Direktiven `%`, `d`, `i`, `o`, `u`, `x`, `X`, `e`, `E`, `f`, `g`, `G`, `c`, `s`, `p`, `n`.

Die meisten der Prozentzeichenfolgen (außer `%%`) „verbrauchen" jeweils ein weiteres Argument aus der Argumentliste hinter dem Parameter `format`. Wenn für die Feldbreite und/oder die Genauigkeit (siehe unten) nur ein `*` angegeben ist, dann wird dafür jeweils ein weiteres Argument vom Typ `signed int` verbraucht. Dieses zusätzliche Argument muß man noch vor dem auszugebenden Wert in die Parameterliste schreiben.

Achtung! Die Direktiven im Formatstring und die restlichen Argumente der Funktion müssen in Typ und Reihenfolge sauber zusammenpassen, sonst produziert die Funktion nur Unsinn!

Erläuterung der Direktiven:

`%%` Mit Hilfe der Formatangabe `%%` wird ein Prozentzeichen ausgegeben. Es wird kein Argument verbraucht.

`%d`, `%i` geben aus der Argumentliste hinter `format` eine Zahl vom Typ `int` dezimal aus.

Wenn eine Genauigkeit angegeben wird, bedeutet diese die mindestens auszugebende Anzahl von Zeichen. Notfalls werden führende Nullen mit ausgegeben. Reicht die Genauigkeit nicht aus, dann werden mehr Zeichen ausgegeben.

Ist keine Genauigkeit angegeben, dann wird der Wert `1` dafür eingesetzt.

`%u` ist wie `%d`, aber die auszugebende Zahl wird als `unsigned int` behandelt.

`%o` gibt ebenfalls eine `unsigned int` aus, aber in oktaler Schreibweise. Mit dem Steuerzeichen `#` wird immer eine führende Null ausgegeben.

`%x`, `%X` geben eine `unsigned int` als sedezimale Zahl (also zur Basis 16) aus. Bei `%x` werden als Ziffern `0`–`9` und `a`–`f` verwendet; bei `%X` stattdessen `0`–`9` und `A`–`F`. Mit dem Steuerzeichen `#` wird vor die Ausgabe der Text `0x` bzw. `0X` gestellt.

`%e`, `%E` geben eine doppelt genaue gebrochene Zahl (`double`) aus, wie etwa `1.234567e-12` oder `-1.234567e+20`. Bei `%E` wird das Zeichen `E` für den Zehnerexponenten ausgegeben. Die Genauigkeit gibt die Anzahl der gewünschten Nachkommastellen an.

Wenn keine Genauigkeit angegeben ist, wird `6` angenommen. Ist die Genauigkeit `0`, dann wird der Dezimalpunkt nur ausgegeben, wenn als Steuerzeichen `#` angegeben wurde.

Es wird immer eine Ziffer vor dem Dezimalpunkt ausgegeben. Diese ist nur dann `0`, wenn die auszugebende Zahl den Wert 0.0 hat.

`%f` gibt ebenfalls eine Gleitkommazahl aus (`double`), aber ohne Exponentialdarstellung. Fehlt die Genauigkeitsangabe, dann wird `6` angenommen.

`%g`, `%G` geben eine Gleitkommazahl (`double`) aus, und zwar in der Form von `%f`, wenn die Zahl in der aktuellen Feldbreite und Genauigkeit ausgegeben werden kann; ansonsten in der Form von `%e` (wenn `%g` angegeben) bzw. `%E` (wenn `%G` angegeben). Eine Genauigkeit von `0` wird wie `1` behandelt. Nachkommastellen werden von rechts her entfernt, soweit sie `0` sind. Dies kann mit dem Steuerzeichen `#` verhindert werden; damit werden auch abschließende Nullen geschrieben.

`%c` gibt ein Argument vom Typ `int` als Zeichen aus.

`%s` verbraucht ein Argument vom Typ `char *`, also ein Zeiger auf `char`. Ab der Stelle, auf die das verbrauchte Argument zeigt, werden Zeichen ausgegeben, bis ein Zeichen den Wert 0 hat, oder die eventuell angegebene Genauigkeit erreicht ist. Die abschließende 0 des Strings wird nicht mit ausgegeben.

`%p` verbraucht einen Zeiger (`void *`) und gibt seinen Wert hexadezimal aus. Wenn das Steuerzeichen `#` angegeben ist, dann wird vor die Ausgabe der Text `0x` gestellt.

`%n` verbraucht einen Zeiger auf eine ganze Zahl (`int *`) und erzeugt keine Ausgabe. Vielmehr wird die Anzahl der bisher ausgegebenen Zeichen ermittelt und dahin geschrieben, wo der verbrauchte Zeiger hinzeigt. Normalerweise wird an diese Stelle eine `int` geschrieben; mit der Längenangabe `h` oder `l` kann man das Schreiben von `short int` oder `long int` erzwingen.

Zwischen dem einleitenden Prozentzeichen und der Direktive darf eine Auswahl der Steuerzeichen `+`, ␣ (*ein Leerzeichen*), `0`, `-`, `#`, *Feldbreite*, `.`*Genauigkeit*, sowie als Hinweis auf die *Objektgröße* der Buchstabe `h` *oder* `l` *oder* `L` stehen.

Gegebenenfalls mehrere dieser Steuerzeichen können in folgender Reihenfolge angegeben werden:

`+` bewirkt, daß bei der Ausgabe von Zahlen ein positives Vorzeichen ausgegeben wird. Standardmäßig würde nur ein negatives Vorzeichen ausgegeben werden.

␣*(Leerz.)* bewirkt, daß positive Zahlen statt mit einem Vorzeichen mit einem Leerzeichen beginnen. Wenn `+` und ␣ angegeben werden, dann gewinnt das `+`.

`0` bewirkt, daß die Ausgabe nicht mit Leerzeichen, sondern mit Nullen auf die angegebene Feldbreite aufgefüllt wird.

`-` bewirkt eine linksbündige Ausgabe innerhalb der Feldbreite. Standard ist rechtsbündig.

`#` bewirkt eine alternative Form der Ausgabe; die Wirkung ist abhängig von der Direktive.

Feldbreite gibt als Dezimalzahl die Anzahl der mindestens auszugebenden Zeichen an. Wird statt einer Zahl ein `*` angegeben, dann wird noch vor dem auszugebenden Argument ein Wert vom Typ `signed int` verbraucht, der für die Feldbreite verwendet wird.

.Genauigkeit kann ebenfalls durch einen `*` mit einem Argument aus der Parameterliste ersetzt werden. Für auszugebende Zeichenketten (Format `%s`) kann man hiermit die maximale Anzahl der zu schreibenden Zeichen an.

Objektgröße gibt die (Speicher-) Größe des auszugebenden Objekts an, und zwar entweder

- `h` bewirkt, daß statt einer `int` eine `short int` verbraucht wird; oder
- `l` statt einer `int` eine `long int` oder
- `L` statt einer `double` eine `long double`.

Leider gibt es keine Direktive, um einen Wert vom Typ `size_t` auszugeben. Dazu muß man mit einem *cast* den auszugebenden Wert in den Typ `unsigned long` verwandeln und dann mit der Direktive `%lu` ausgeben.

Die Funktion `printf()` gibt die Anzahl der ausgegebenen Zeichen zurück. Tritt ein Fehler auf, dann wird eine negative Zahl zurückgegeben und `errno` gesetzt.

→**fprintf()** (Seite 193), **sprintf()** (Seite 223), **vprintf()** (Seite 239),
→**vfprintf()** (Seite 238), **vsprintf()** (Seite 238), **scanf()** (Seite 218),
→**puts()** (Seite 213), **putchar()** (Seite 212)

putc()

```
#include <stdio.h>

int putc( int zeichen, FILE *datei );
```

gibt das Zeichen `zeichen` auf `datei` aus.

Eigentlich gleichbedeutend mit `fputc()`. Aber bei manchen Systemen als Makro definiert und deshalb unsicher.

Rückgabewert ist das ausgegebene Zeichen, oder `EOF`, wenn ein Fehler auftritt. Dann werden auch `errno` und der Fehlerindikator in `datei` gesetzt.

→**fputc()** (Seite 194), **putchar()** (Seite 212),
→**puts()** (Seite 213)

putchar()

```
#include <stdio.h>

int putchar( int zeichen );
```

gibt das Zeichen `zeichen` auf der Standardausgabe aus.

Gleichbedeutend mit `putc( c, stdout )`.

Rückgabewert ist das ausgegebene Zeichen, oder `EOF`, wenn ein Fehler auftritt. Dann wird auch `errno` gesetzt.

→**fputc()** (Seite 194), **putc()** (Seite 212), **puts()** (Seite 213)

puts()

```
#include <stdio.h>

int puts( char *s );
```

gibt den String, der an der Stelle `s` beginnt, und einen Zeilenvorschub auf die Standardausgabe aus.

Rückgabewert ist 0, wenn alles klar ging. Im Fehlerfall wird ein Wert ungleich 0 zurückgegeben und `errno` sowie der Fehlerindikator der Standardausgabe werden gesetzt.

→**printf()** (Seite 210), **gets()** (Seite 199), **fputs()** (Seite 194), **putchar()** (Seite 212)

qsort()

```
#include <stdlib.h>

void qsort( void *feldanfang
            size_t elemente,
            size_t elementgroesse,
            int (*vergleichsfunktion)( const void *zeiger1,
                                       const void *zeiger2 )
          );
```

sortiert ein beliebiges Feld im Speicher, wenn folgende Bedingungen erfüllt sind:

- Das Feld beginnt an der Stelle `feldanfang` im Speicher.
- Das Feld hat `elemente` Elemente jeweils der Größe `elementgroesse`.
- `vergleichsfunktion` zeigt auf eine Funktion, die zwei Zeiger auf je ein Feldelement entgegennimmt und einen negativen `int`-Wert liefert, wenn der erste Parameter auf ein Element mit kleinerem Suchschlüssel zeigt als der zweite Parameter, 0 liefert, wenn beide Parameter auf Elemente mit gleich großem Suchschlüssel zeigen und sonst einen positiven Wert liefert.

Gemäß den Vergleichskriterien der Funktion `(*vergleichsfunktion)()` ist das Feld nach dem Aufruf von `qsort()` sortiert.

Liefern mehrere Elemente keinen Unterschied beim Vergleich, dann ist ihre Reihenfolge untereinander unbestimmt und nicht unbedingt ihre ursprüngliche.

Wenn die einzelnen Feldelemente groß sind, kann man meist Rechenzeit sparen, indem man ein Feld mit Zeigern auf die einzelnen Feldelemente anlegt und dieses Zeigerfeld sortieren läßt. Die Vergleichsfunktion muß natürlich eine Dereferenzierung mehr besitzen für jeden der Parameter.

Beispielaufruf:

```
/* fuqsort.c 30. 7.95 kw
 *
 * Vergleichsfunktion fuer zwei int-Werte.
 * Liefert einen Wert gleich 0, wenn die Elemente *p1 und *p2
 * gleich sind, kleiner als 0, wenn *p1 kleiner als *p2, sonst
 * groesser 0.
 */
int verg( const void *p1, const void *p2 )
{   return *(int *)p1 - *(int *)p2;
}

/* Testprogramm fuer qsort().
 */
int main()
{
  int   feld[10] = { 45, 89, 12, 36, 78, 1, 2, 41, 74, 85 };
  int   i;

  /* unsortiert ausgeben:
   */
  for( i = 0; i<10; i++ )
    printf( "%d ", feld[i] );
  printf( "\n" );

  /* sortieren lassen:
   */
  qsort( feld, 10, sizeof(int), verg );

  /* sortiert ausgeben:
   */
  for( i = 0; i<10; i++ )
    printf( "%d ", feld[i] );
  printf( "\n" );

  return;
}
```

Ausgabe davon:

```
45 89 12 36 78 1 2 41 74 85
1 2 12 36 41 45 74 78 85 89
```

Zum Parameter `verg` beim Aufruf von `qsort()` siehe auch die Bemerkung (bezüglich Adresse von Funktionen) auf Seite 44.

Ein weiteres Beispiel findet man auf Seite 181.

→**bsearch()** (Seite 181)

raise()

```
#include <signal.h>

int raise( int welches_signal );
```

erzeugt eine Ausnahmebehandlung, in der an das laufende Programm ein Signal mit dem Wert `welches_signal` gesandt wird.

`welches_signal` sollte einer der in signal.h definierten Werte sein, etwa `SIGFPE` (siehe Seite 221).

Mit der Funktion **signal()** (Seite 221) kann vorher festgelegt werden, wie auf bestimmte Signale zu reagieren ist.

→**signal()** (Seite 221)

rand()

```
#include <stdlib.h>

int rand( void );
```

erzeugt eine „Zufallszahl“ im Bereich von 0 bis `RAND_MAX`.

Dies ist bei jedem Aufruf eine weitere Zahl aus einer Zahlenfolge, deren Startwert mit `srand()` gesetzt werden kann. Sonst wird dieser Startwert bei jedem Programmstart auf 0 gesetzt.

Ein Programm erhält dadurch bei jedem Lauf die gleiche Folge von „Zufallszahlen“:

```
/* furand1.c 30. 7.95 kw
 */

#include <stdlib.h>

int main()
{
  int     i;

  for( i=0; i<10; i++ )
    printf( "%d\n", rand() );
}
```

Die Ausgabe des obigen Programms ist bei jedem Programmstart gleich.

Die immer wieder gleiche Folge von Zufallszahlen ist zur Fehlersuche manchmal ganz nützlich, da dadurch die Daten immer gleich sind. In vielen praktischen Fällen ist das aber nicht erwünscht; dann muß man den Startwert bewußt bei jedem Programmstart auf einen anderen Wert setzen. Dazu bieten sich Datum und Uhrzeit an:

```
/* furand2.c 30. 7.95 kw
 */

#include <stdlib.h>
```

```
#include <time.h>

int main()
{
  int         i;
  time_t      t;
  struct tm  *tp;


  /* Datum und Uhrzeit holen:
   */
  t  = time( NULL );
  tp = localtime( &t );

  /* den Startwert fuer die Zufallszahlen
   * in Abhaengigkeit von der Zeit setzen:
   */
  srand(   tp->tm_sec
         + tp->tm_min
         + tp->tm_hour
         + tp->tm_yday
         );


  for( i=0; i<10; i++ )
    printf( "%d\n", rand() );
}
```

→**srand()** (Seite 223)

realloc()

```
#include <stdlib.h>

void *realloc( void *alterblock, size_t wieviel )
```

dient dazu, einen Speicherblock zu allokieren (wie `malloc()`), oder einen bereits allokierten Speicherblock freizugeben (wie `free()`), oder einen allokierten Speicherblock in der Größe zu verändern.

In Abhängigkeit seiner Parameter verhält sich die Funktion `realloc()` folgendermaßen:

Wenn `alterblock == NULL`,

dann: `realloc()` reagiert genau wie `malloc( wieviel )`, versucht also, einen Speicherblock der Größe `wieviel` zu allokieren.

ansonsten: `alterblock` muß auf einen bereits früher mit `malloc()` oder `calloc()` reservierten Speicherblock zeigen. `realloc()` versucht dann, den Speicherblock auf die neue Größe `wieviel` zu ändern. Ist `wieviel==0`, dann wird der früher allokierte Block freigegeben (wie bei `free( alterblock )`). Nach der Größenänderung kann der Block an einer anderen Stelle im Speicher liegen. Dann wird der alte Blockinhalt automatisch an die neue Stelle kopiert (maximal jedoch `wieviel` Bytes).

Rückgabewert ist ein Zeiger auf den Anfang des reservierten Blocks, wenn alles gut ging, oder der `NULL`-Zeiger, wenn kein Block ausreichender Größe zu haben ist. Ist `wieviel==0`, dann wird der Block freigegeben und der Rückgabewert ist `NULL`.

→**free()** (Seite 195), **malloc()** (Seite 206), **calloc()** (Seite 184)

remove()

```
#include <stdio.h>

int remove( const char *dateiname );
```

entfernt (löscht) eine Datei von einem Massenspeicher.

Rückgabe ist im Erfolgsfall der Wert 0. Ansonsten wird ein Wert ungleich 0 geliefert und `errno` gesetzt.

→**rename()** (Seite 217), **tmpfile()** (Seite 234), **tmpnam()** (Seite 234)

rename()

```
#include <stdio.h>

int rename( const char *altername, const char *neuername );
```

benennt eine Datei um. Dies ist auch möglich, wenn die Datei gerade geöffnet ist.

Eine Datei mit dem Namen `neuername` darf vorher nicht existieren.

Rückgabe ist im Erfolgsfall der Wert 0. Ansonsten wird ein Wert ungleich 0 geliefert und `errno` gesetzt.

→**remove()** (Seite 217)

rewind()

```
#include <stdio.h>

void rewind( FILE *datei );
```

spult eine Datei an den Anfang zurück und löscht gegebenenfalls den Fehler- und den Dateiendeindikator der Datei. Nach `rewind()` darf bei Dateien im Änderungsmodus zwischen Lese- und Schreibzugriff gewechselt werden.

→**fopen()** (Seite 192), **fseek()** (Seite 197), **ftell()** (Seite 198), **fsetpos()** (Seite 198), **fgetpos()** (Seite 190)

scanf()

```
#include <stdio.h>

int scanf( const char *format, ... );
```

liest von der Standardeingabe Zeichen und führt gemäß den Direktiven in dem String ab `*format` Datenumwandlungen aus. Dabei werden von den Direktiven eventuell weitere Parameter verbraucht, deren Typ und Reihenfolge von den Direktiven in `format` abhängen. Da diese zusätzlichen Parameter angeben, wo die Ergebnisse der Umwandlungen im Speicher abzulegen sind, sind alle diese zusätzlichen Parameter von einem Zeigertyp.

Die Folge aus Eingabe und Umwandlungen wird solange fortgesetzt, bis das Ende von `format` erreicht ist, oder bis die eingelesenen Zeichen nicht zu den Direktiven passen. In diesem Fall wird die Abarbeitung vorzeitig abgebrochen.

Zurückgegeben wird die Anzahl der tatsächlich umgewandelten und zugewiesenen Werte oder `EOF`, wenn ein Dateifehler auftritt.

Der Formatstring besteht aus einer Folge von Direktiven. Jedes Zeichen außer dem Prozentzeichen `%` bewirkt, daß ein Zeichen von der Eingabe gelesen wird und mit dem Zeichen aus dem Formatstring verglichen wird. Wenn das gelesene Zeichen nicht identisch mit mit dem in `format`, dann wird `scanf()` abgebrochen.

Alle anderen Direktiven beginnen mit einem Prozentzeichen. Danach können in dieser Reihenfolge folgen (*müssen aber nicht!*):

`*` Dies bedeutet, daß das Ergebnis der Umwandlung aus dieser Direktive keinem Wert zugewiesen werden soll und dadurch auch kein Argument verbraucht wird.

Weiterhin beeinflußt die Anzahl der so verarbeiteten Eingaben *nicht* den Rückgabewert der Funktion.

Die Eingabe wird also einfach überlesen.

Feldbreite gibt (dezimal) an, wieviele Zeichen für die Umwandlung maximal gelesen werden sollen.

Objektgröße ist ein Zeichen, welches die Größe des umgewandelten Ergebnisses im Speicher angibt und damit den Typ des zugeordneten Zeigers in der Parameterliste beschreibt:

- `h` gibt an, daß eine ganzzahlige Umwandlung den Typ `short int` haben soll.
- `l` analog `long int`; bei Gleitkommakonversion bedeutet `l` eine Umwandlung nach `double` statt nach `float`.
- `L` bedeutet `long double`.

Wenn dieses Zeichen fehlt, dann wird für ganzzahlige Umwandlung der Datentyp `int` (`signed` oder `unsigned`) angenommen und für Gleitkommazahlen der Typ `float`.

Die Direktive endet mit einem Zeichen, das die Art der Umwandlung angibt:

`%d` führt eine Umwandlung der gelesenen Zeichen von dezimaler Schreibweise nach `int` durch.

`%i` wandelt eine Zeichenfolge mit einer vorzeichenbehafteten ganzen Zahl um, wie sie auch von `strtol()` mit der Basis 0 durchgeführt wird. Die Zahlenbasis der Umwandlung wird dabei von den ersten Zeichen festgelegt. Siehe `strtol()`.

`%o` wandelt eine vorzeichenlose oktale Zeichenfolge (Basis 8) um. Siehe `strtoul()` mit Basis 8.

`%u` wandelt eine vorzeichenlose Ziffernfolge (dezimal, also Basis 10) um wie `strtoul()` mit Basis 10.

`%x`, `%X` verwandeln eine sedezimale Ziffernfolge wie `strtoul()` mit Basis 16.

`%e`, `%E` ,

`%f`, `%F` ,

`%g`, `%G` wandeln die Zeichenfolge einer Gleitkommazahl wie `strtod()`.

`%c` wandelt eine Zeichenfolge mit der Länge der Feldbreite, wenn angegeben; sonst ein Zeichen. Hier werden ausnahmsweise führende *white-space*-Zeichen nicht übersprungen. Das zugehörige Argument muß ein Zeiger auf einen Speicherbereich sein, der groß genug ist für alle gelesenen Zeichen. Es wird keine abschließende Null angehängt.

`%s` liest eine Folge von Zeichen, die nicht Zwischenraumzeichen sind. Der zugehörige Parameter muß auf einen Speicherbereich zeigen, der groß genug ist für die gelesenen Zeichen und die abschließende `\0`, die automatisch angehängt wird.

Wenn eine Feldbreite angegeben ist, dann werden maximal soviele Zeichen eingelesen und in den Speicherbereich kopiert; wie sonst auch wird dann eine abschließende Null angehängt. Deshalb sollte der Speicherbereich ein Byte größer sein als die Feldbreite!

`%[` beginnt in `format` eine Zeichenfolge, die von `]` beendet wird. Im eingelesenen Strom von Zeichen werden alle Zeichen in den Speicher übertragen ab der Stelle, auf die der zugehörige Parameter aus der Parameterliste zeigt. Dieser Speicherbereich muß Platz bieten für alle eingelesenen Zeichen (maximal die Feldbreite) plus 1 für die abschließende `\0`, die automatisch angefügt wird. Es wird übertragen, solange alle gelesenen Zeichen in dem Klammerpaar `[...]` enthalten sind; das erste andere Zeichen wird nicht übertragen.

Ist das erste Zeichen nach dem `[` ein `^`, dann wird die Wirkung umgedreht: alle eingelesenen Zeichen, die nicht in `format` zwischen `[^` und `]` vorkommen, werden übertragen.

Um das Zeichen `]` selbst in die Zeichenfolge aufzunehmen, muß es direkt nach dem Zeichen `[` auftreten bzw. nach der Folge `[^`. Nur dann beendet dieses `]` nicht die Direktive in `format`, sondern erst das nächste auftretende `]`.

`%p` wandelt eine hexadezimale Zeichenfolge `0x...` um in einen Zeiger. Das zugehörige Argument muß also vom Typ *Zeiger auf Zeiger auf irgendwas* (`void **`) sein.

`%n` liest keine Eingabezeichen; aber die Anzahl der bisher eingelesenen Zeichen wird als `unsigned int` dahin geschrieben, wo der zugehörige Zeiger aus der Parameterliste hinzeigt. Dieser Vorgang erhöht nicht den Rückgabewert von `scanf()`.

%% paßt zu einem % aus der Eingabe. Wie sonst auch jede Direktive, die nicht mit % beginnt, wird kein Parameter verbraucht. Diese Direktive sollte nur aus den beiden Zeichen %% bestehen; die sonst wahlweise möglichen Zeichen dazwischen dürfen nicht genutzt werden.

Die Umwandlung geschieht in folgender Weise:

Alle Zwischenraumzeichen (*white space*, siehe **isspace()** (Seite 202)) werden übersprungen.

Von der Eingabe werden Zeichen gelesen, bis die Feldbreite erreicht ist, oder keine Eingabezeichen mehr verfügbar sind, oder bis ein Zeichen nicht zur aktuellen Umwandlung paßt. In letzterem Fall wird das gelesene Zeichen nicht verbraucht und steht für die nächste Umwandlung zur Verfügung.

Je nach dem letzten Buchstaben der Direktive (siehe oben) wird die Folge der Eingabezeichen in den entsprechenden Datentyp umgewandelt. Dabei kann ein Überlauf auftreten, der das Ergebnis verfälscht und nicht weiter berücksichtigt wird.

Wenn das Zeichen `*` nicht zu Beginn der Direktive stand, dann wird aus der Parameterliste ein Zeiger verbraucht und das Ergebnis an der betreffenden Stelle im Speicher abgelegt. Jede solche Zuweisung erhöht den späteren Rückgabewert um eins.

Achtung! Die Direktiven im Formatstring und die restlichen Argumente der Funktion müssen in Typ und Reihenfolge sauber zusammenpassen, sonst produziert die Funktion nur Unsinn!

Ebenso wie bei `printf()` gibt es für den Datentyp `size_t` keine direkt passende Direktive. Dazu muß man den Umweg über eine Variable vom Typ `unsigned long` nehmen.

→**strtol()** (Seite 231), **printf()** (Seite 210), **sscanf()** (Seite 224),
→**fscanf()** (Seite 196)

setbuf()

```
#include <stdio.h>

void setbuf( FILE *datei, char *puffer );
```

setzt den Puffer für die angegebene Datei an die Stelle `puffer`.

Der Speicherbereich ab diesem Zeigerwert muß mindestens `BUFSIZ` Bytes groß sein und muß existieren, solange die Datei geöffnet bleibt.

Wenn `puffer` den Wert `NULL` hat, dann wird die Ein- und Ausgabe für diese Datei nicht gepuffert.

Diese Funktion muß zwischen dem Öffnen der Datei und dem ersten Zugriff aufgerufen werden oder gar nicht.

`setbuf( datei, puffer )` ist identisch mit
`setvbuf( datei, puffer, _IOFBF, BUFSIZ )` oder (wenn `puffer` `NULL` ist) mit
`setvbuf( datei, puffer, _IONBF, 0 )`.

→**fopen()** (Seite 192), **setvbuf()** (Seite 221)

setjmp()

```
#include <setjmp.h>

int setjmp( jmp_env umgebung );
```

speichert die aktuelle Situation des Prozessors (*program counter*, *stack pointer*, alle wichtigen Register etc.) in `umgebung` und gibt den Wert 0 zurück.

Bei einem späteren Aufruf von `longjmp( umgebung, wert )` wird ebenfalls die Programmausführung da fortgesetzt, wo sie auch nach dem Aufruf von `setjmp()` fortgeführt wird. Nur wird nach der Ausführung von `longjmp()` nicht 0 geliefert, sondern der zweite Parameter von `longjmp()`.

Deshalb sollte dieser nie 0 sein; sonst kann der Aufrufer von `setjmp()` nicht entscheiden, ob der Rücksprung aus `setjmp()` tatsächlich von `setjmp()` kommt oder von einem späteren Aufruf von `longjmp()`.

Ein Beispiel steht auf Seite 205. Die dort stehende Warnung bitte beherzigen!

→**longjmp()** (Seite 205)

setvbuf()

```
#include <stdio.h>

int setvbuf( FILE *datei,
             char *puffer,
             int wie_denn_nun,
             size_t wieviel    );
```

gibt die gewünschte Pufferung für eine Datei an.

`setvbuf()` muß zwischen dem Öffnen der Datei und dem ersten Zugriff ausgeführt werden oder gar nicht.

`wie_denn_nun` muß einer der Werte: `_IOFBF`, `_IOLBF`, `_IONBF` sein. Dementsprechend wird die Datei voll (blockweise), zeilenweise oder gar nicht gepuffert.

Außer bei `_IONBF` muß `puffer` auf einen Speicherbereich zeigen, der `wieviel` Byte lang ist und solange erhalten bleibt, wie die Datei geöffnet ist.

Ist `puffer` gleich `NULL`, dann wird von `setvbuf()` selbständig ein ausreichender Puffer beschafft (mit `malloc()`), der beim Schließen der Datei auch wieder freigeben wird.

Zurückgegben wird 0, wenn alles gut ging; ansonsten ein Wert ungleich 0. Dann wird `errno` entsprechend gesetzt.

→**setbuf()** (Seite 220)

signal()

```
#include <signal.h>

void (*signal( int welches_signal,
               void (*neue_funktion)( int )
             )
     )( int );
```

gibt an, wie in Zukunft auf Signale mit dem Wert `welches_signal` reagiert werden soll.

`signal()` hat zwei Parameter:

- eine `int`, die angibt, um welches der in Zukunft möglicherweise auftretenden Signale es geht.
- einen Zeiger auf eine Funktion, die wiederum einen `int`-Parameter erhält und nichts `void` zurückliefert.

`welches_signal` sollte einer der folgenden Werte sein:

- `SIGABRT` Programmabbruch, z.B. wegen Aufruf von `abort()`.
- `SIGFPE` Fehler bei Gleitkommarechnung.
- `SIGILL` unzulässiger Funktionsaufruf.
- `SIGINT` Eintreffen eines interaktiven Signals.
- `SIGSEGV` Unzulässiger Speicherzugriff.
- `SIGTERM` Programmende gefordert.

`neue_funktion` muß entweder auf eine Funktion zeigen, die in Zukunft auf das entsprechende Signal reagieren soll (der Signaltyp, also `welches_signal`, wird ihr dann als Parameter übergeben), oder `neue_funktion` muß einen der Werte `SIG_IGN` bzw. `SIG_DFL` haben. Dann wird beim Auftreten eines solchen Signals selbiges ignoriert bzw. die Standardreaktion findet statt, die auch bei Programmstart voreingestellt ist.

Zurückgegeben wird ein Zeiger auf den bisherigen Empfänger eines solchen Signals.

Wirkliches Weitergeben von Signalen geschieht eigentlich nur auf Systemen mit Multitasking, also nicht auf kleinen Rechnern mit MSDOS oder ähnlichen Betriebssystemen.

Ein Signal kann jedoch auch mit `raise()` vom Programm selbst erzeugt und an sich selbst geschickt werden.

→**raise()** (Seite 215)

sin()

```
#include <math.h>

double sin( double phi );
```

liefert den sin(`phi`) (`phi` ist im Bogenmaß anzugeben).

Ist `phi` betragsmäßig zu groß, dann kann das Ergebnis ungenau sein, ohne daß `errno` gesetzt wird.

→**asin()** (Seite 178), **cos()** (Seite 186), **tan()** (Seite 232), **sinh()** (Seite 223)

sinh()

```
#include <math.h>

double sinh( double phi );
```

liefert den sinh(phi).

Ist phi betragsmäßig zu groß, dann wird der Wert HUGE_VAL zurückgegeben und errno auf ERANGE gesetzt.

→**sin()** (Seite 222), **cosh()** (Seite 186), **tanh()** (Seite 232)

sprintf()

```
#include <stdio.h>

int sprintf( char *string, const char *format, ... );
```

ist identisch mit `printf()`, mit dem Unterschied, daß die Ausgabe nicht nach stdout geschrieben wird, sondern stattdessen in den Speicher ab der Stelle string. Dieser Parameter muß also auf einen ausreichend großen Speicherbereich zeigen. Der geschriebene String wird automatisch mit \0 abgeschlossen.

Zur Beschreibung der weiteren Parameter: siehe `printf()`.

→**printf()** (Seite 210), **sscanf()** (Seite 224)

sqrt()

```
#include <math.h>

double sqrt( double wert );
```

berechnet $\sqrt{\texttt{wert}}$ und gibt diese Zahl zurück.

Wenn wert negativ ist, dann wird 0.0 zurückgegeben und errno auf EDOM gesetzt.

→**pow()** (Seite 209)

srand()

```
#include <stdlib.h>

void srand( int startwert );
```

setzt den Startwert des Zufallszahlengenerators für zukünftige Aufrufe von `rand()`.

Dieser Startwert wird bei jedem Programmstart auf 0 gesetzt. Mit `srand( 0 )` und einer Folge von `rand()`-Aufrufen kann man also jeweils wieder die gleiche Zahlenfolge wie bei Programmstart erhalten.

→**rand()** (Seite 215)

sscanf()

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

```
int sscanf( const char *string, const char *format, ... );
```

ist identisch mit `scanf()`, außer dem zusätzlichen ersten Parameter. `sscanf()` liest nicht von `stdin`, sondern aus dem Speicherbereich, der bei `string` beginnt.

Zur Beschreibung der weiteren Parameter: siehe `scanf()`.

→**scanf()** (Seite 218), **sprintf()** (Seite 223)

strcat()

```
#include <string.h>
```

```
char *strcat( char *woran, char *woher );
```

kopiert an das Ende des Strings `woran` alle Zeichen ab `*woher`, wobei die abschließende `\0` des Strings `woran` vom ersten Zeichen aus `woher` überschrieben wird. Das Resultat ist ein String, der ab `woran` steht und hinter den bisherigen Zeichen von `woran` die Zeichen ab `woher` enthält. Dahinter folgt eine abschließende Null `\0`.

Achtung! Der Zeiger `woran` muß auf einen Speicherbereich zeigen, der genügend Platz für alle Zeichen aus `woran` plus alle Zeichen aus `woher` plus 1 (für die Null) enthält.

Zurückgegeben wird ein Zeiger mit dem Wert `woran`.

→**strncat()** (Seite 228), **memcpy()** (Seite 207), **strcpy()** (Seite 225)

strchr()

```
#include <string.h>
```

```
char *strchr( const char *wo, int was );
```

sucht in dem String ab `*wo` nach dem Zeichen `(char)was` und liefert einen Zeiger auf das erste Vorkommen, bzw. `NULL`, wenn nicht gefunden. Die abschließende `\0` wird in die Suche noch mit eingeschlossen, das heißt man kann mit `strchr( string, 0 )` nach dem Stringende suchen.

→**strlen()** (Seite 227), **memchr()** (Seite 207), **strstr()** (Seite 229), **strrchr()** (Seite 229)

strcmp()

```
#include <string.h>
```

```
int strcmp( const char *str1, const char *str2 );
```

vergleicht zwei Strings `str1` und `str2` und liefert einen Wert größer als 0, wenn in der ersten Position, an der sich die beiden Strings unterscheiden, in `str1` ein (alphabetisch) größeres Zeichen steht als in `str2`, und liefert einen Wert kleiner als 0 bei umgekehrten Verhältnissen. Sind die beiden Strings gleichlang und bis zur abschließenden `\0` identisch, dann wird 0 zurückgegeben.

Die Zeichen in den Strings werden dabei immer als `unsigned char` betrachtet.

→**strncmp()** (Seite 228), **memcmp()** (Seite 207)

strcpy()

```
#include <string.h>

char *strcpy( char *wohin, char *woher );
```

kopiert alle Zeichen ab `*woher` bis zur ersten auftretenden `\0` (einschließlich) in den Speicher ab der Stelle `wohin`.

Achtung! Der Zeiger `wohin` muß auf einen Speicherbereich zeigen, der genügend Platz für alle Zeichen aus `woher` plus 1 (für die Null) enthält.

→**strncpy()** (Seite 228), **memcpy()** (Seite 207), **strcat()** (Seite 224)

strcspn()

```
#include <string.h>

size_t strcspn( const char *worin, const char *welchezeichennicht );
```

bestimmt die Länge des Anfangsstücks des Strings `worin`, das nur aus Zeichen besteht, die nicht auch in `welchezeichennicht` vorkommen.

Wenn `welchezeichennicht` leer ist oder keine Zeichen enthält, die auch in `worin` vorkommen, dann liefert `strcspn()` die Länge von `worin`.

→**strspn()** (Seite 229)

strerror()

```
#include <string.h>

char *strerror( int fehlernummer );
```

liefert einen Zeiger auf einen String, der eine Beschreibung des Fehlers mit der Nummer `fehlernummer` enthält.

`fehlernummer` muß dazu einer der Werte sein, die in errno.h für die Variable `errno` definiert sind. Ansonsten kann es sein, daß `strerror()` einen Zeiger auf einen leeren String liefert.

Die Fehlerbeschreibung darf nicht modifiziert werden.

Achtung! Bei einigen Unix-Versionen gibt es eine Funktion mit dem gleichen Namen, aber einer anderen Funktionsweise!

→**perror()** (Seite 209)

strftime()

```
#include <time.h>

size_t strftime( char *wohin,
                 size_t wieviele_maximal,
                 const char *format,
                 const struct tm *ortszeit    );
```

erzeugt einen String ab der Stelle `wohin` mit maximal `wieviele_maximal` Zeichen (einschließlich der abschließenden `\0`), der aus `format` gebildet wird. Alle Zeichen aus `format` werden nach `wohin` kopiert, mit Ausnahme von bestimmten Direktiven, die in den erzeugten String `wohin` Werte aus `*ortszeit` einsetzen.

An Direktiven stehen zur Verfügung:

- `%a` abgekürzter Name des Wochentags
- `%A` Name des Wochentags
- `%b` abgekürzter Monatsname
- `%B` Monatsname
- `%c` Datum und Uhrzeit
- `%d` Tag des Monats (`01--31`)
- `%H` Stunde (`00--23`)
- `%I` Stunde (`00--12`)
- `%j` Tag im Jahr (`001--366`)
- `%m` Monat im Jahr (`01--12`)
- `%M` Minute (`00--59`)
- `%p` `AM` oder `PM`
- `%S` Sekunde (`00--59`)
- `%U` Wochennummer mit Sonntag als erstem Tag der Woche (`00--53`)
- `%w` Wochentag mit 0 als Sonntag (`0--6`)
- `%W` Wochennummer mit Montag als erstem Tag der Woche (`00--53`)
- `%x` Datum
- `%X` Uhrzeit
- `%y` Jahr ohne Jahrhundert (`00--99`)
- `%Y` Jahr mit Jahrhundert
- `%Z` Name der Zeitzone, oder nichts, falls die Zeitzone nicht feststellbar ist. Dieser Wert wird aus der Environmentvariablen `TZ` bestimmt.
- `%%` Ein Prozentzeichen wird ausgegeben.

Zurückgegeben wird 0, wenn der String nicht innerhalb der angegebenen Zeichenzahl gebildet werden kann; ansonsten die Anzahl der erzeugten Zeichen (ohne die `\0`).

Das folgende Beispiel gibt das aktuelle Datum aus:

```
/* fustrft.c 30. 7.95 kw
 */

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

#define LAENGE  25

int main( int argc, char **argv )
{
  char                  string[LAENGE];
  struct tm   ortszeit;
  time_t                zeitgepackt;

  /* Holen des gepackten Zeitformats:
   */
  zeitgepackt = time( NULL );
  /* Entpacken in eine struct tm:
   */
  ortszeit = *localtime( &zeitgepackt );
  /* Tag, Monat und Jahr in einen String
   * schreiben:
   */
  strftime( string,
            LAENGE,
            "Heute ist der %d.%m.%Y\n",
            &ortszeit
            );
  /* Ausgabe des Strings:
   */
  puts( string );

  return EXIT_SUCCESS;
}
```

→**time()** (Seite 232), **asctime()** (Seite 177), **localtime()** (Seite 204)

strlen()

```
#include <string.h>

size_t strlen( const char *string );
```

bestimmt die Länge des Strings `string` bis auf die abschließende `\0`.

→**strchr()** (Seite 224)

strncat()

```
#include <string.h>

char *strncat( char *woran, char *woher, size_t wieviele_maximal );
```

kopiert an das Ende des Strings `woran` alle Zeichen ab `*woher`, wobei die abschließende `\0` des Strings `woran` vom ersten Zeichen aus `woher` überschrieben wird. Das Resultat ist ein String, der ab `*woran` steht und hinter den bisherigen Zeichen von `woran` die Zeichen ab `*woher` enthält. Dahinter folgt eine abschließende Null `\0`.

Die Anzahl der kopierten Zeichen ist maximal `wieviele_maximal` Zeichen ohne die abschließende `\0`. Darüber hinaus gehende Zeichen werden nicht kopiert.

Achtung! Der Zeiger `woran` muß auf einen Speicherbereich zeigen, der für (*die alte Länge von woran* + `wieviele_max` + 1) Zeichen Platz bietet.

Die Strings `woher` und `wohin` dürfen sich nicht überschneiden.

Zurückgegeben wird ein Zeiger mit dem Wert `woran`.

→**strcat()** (Seite 224), **memcpy()** (Seite 207), **strncpy()** (Seite 228)

strncmp()

```
#include <string.h>

int strncmp( const char *str1, const char *str2, size_t wieviele_maximal );
```

vergleicht zwei Strings `str1` und `str2` und liefert einen Wert größer als 0, wenn in der ersten Position, an der sich die beiden Strings unterscheiden, in `str1` ein (alphabetisch) größeres Zeichen steht als in `str2`, und liefert einen Wert kleiner als 0 bei umgekehrten Verhältnissen. Sind die beiden Strings gleichlang und bis zur abschließenden `\0` identisch, dann wird 0 zurückgegeben.

Es werden höchstens `wieviele_maximal` Zeichen verglichen. Sind beide Strings länger und in den ersten `wieviele_maximal` Zeichen gleich, dann wird ebenfalls 0 zurückgegeben.

Die Zeichen in den Strings werden dabei immer als `unsigned char` betrachtet.

→**strcmp()** (Seite 224), **memcmp()** (Seite 207)

strncpy()

```
#include <string.h>

char *strncpy( char *wohin, char *woher, size_t wieviele );
```

kopiert alle Zeichen ab `woher` bis zur ersten auftretenden `\0` (einschließlich) in den Speicher ab der Stelle `wohin` und füllt bis zur Länge `wieviele` mit Nullen auf, wenn `woher` kürzer ist als `wieviele` Zeichen. Ansonsten werden `wieviele` Zeichen kopiert und keine Null angehängt.

Achtung! Der Zeiger `wohin` muß auf einen Speicherbereich zeigen, der genügend Platz für `wieviele` Zeichen enthält.

→**strcpy()** (Seite 225), **memcpy()** (Seite 207), **strcat()** (Seite 224)

strpbrk()

```
#include <string.h>

char *strpbrk( const char *worin, const char *welche );
```

liefert einen Zeiger auf das erste Zeichen in dem String `worin`, das auch in dem String `welche` vorkommt. Die abschließenden Nullen werden nicht in die Suche mit einbezogen. Wird kein solches Zeichen gefunden, dann liefert die Funktion `NULL` zurück.

→**strstr()** (Seite 229), **strspn()** (Seite 229), **strchr()** (Seite 224),
→**strtok()** (Seite 230)

strrchr()

```
#include <string.h>

char *strrchr( const char *wo, int was );
```

sucht in dem String ab `wo` nach dem Zeichen `(char)was` und liefert einen Zeiger auf das letzte Vorkommen, bzw. `NULL`, wenn nicht gefunden. Die abschließende `\0` wird in die Suche mit eingeschlossen, das heißt man kann mit `strrchr( string, 0 )` nach dem Stringende suchen.

→**strlen()** (Seite 227), **memchr()** (Seite 207), **strstr()** (Seite 229),
→**strchr()** (Seite 224)

strspn()

```
#include <string.h>

size_t strspn( const char *worin, const char *welchezeichen );
```

bestimmt die Länge des Anfangsstücks des Strings `worin`, das nur aus Zeichen besteht, die auch in `welchezeichen` vorkommen.

Die abschließenden Nullen werden nicht mit durchsucht.

→**strcspn()** (Seite 225)

strstr()

```
#include <string.h>

char *strstr( const char *worin, const char *welcher );
```

sucht in dem String `worin` nach der Zeichenfolge `welcher` und liefert einen Zeiger auf den Anfang in `worin` bzw. `NULL`, wenn `welcher` nicht in `worin` enthalten ist. Die abschließende Null von `welcher` wird nicht mit einbezogen.

→**strchr()** (Seite 224), **strpbrk()** (Seite 229), **strspn()** (Seite 229),
→**strcspn()** (Seite 225)

strtod()

```
#include <stdlib.h>

double strtod( const char *string, char **endezeiger );
```

verwandelt den String ab `string` in eine Gleitkommazahl, die zurückgegeben wird. Dabei werden aus `string` solange Zeichen gelesen, bis das Ende von `string` erreicht ist oder ein ungültiges Zeichen gelesen wird. Der dabei errechnete Zahlenwert wird zurückgegeben. Wenn `endezeiger` ungleich `NULL` ist, dann wird an der Stelle `*endezeiger` im Speicher ein Zeiger auf das erste Zeichen hinterlegt, das nicht mehr mit umgewandelt werden konnte. War bereits das erste Zeichen ungültig, dann ist der hinterlegte Zeiger der Wert von `string`.

Kann kein Zeichen umgewandelt werden, dann wird 0.0 zurückgegeben. Ist der umzuwandelnde Wert zu groß, dann wird `+HUGE_VAL` oder `-HUGE_VAL` zurückgegeben und `errno` auf `ERANGE` gesetzt.

→**atof()** (Seite 180), **sscanf()** (Seite 224), **strtol()** (Seite 231),
→**strtoul()** (Seite 231)

strtok()

```
#include <string.h>

char *strtok( char *string, const char *trennzeichen );
```

teilt einen String `string` in einzelne Bruchstücke (*token*) gemäß der Trennzeichen im String `trennzeichen`.

Für jedes Bruchstück erfolgt ein Aufruf von `strtok()`, wobei jeweils ein Zeiger auf den Anfang des Tokens geliefert wird oder `NULL`, wenn der String keine weiteren Token liefert.

Beim ersten Aufruf übergibt man den zu zerlegenden String als ersten Parameter. Die Menge der Trennzeichen übergibt man in einem String als zweiten Parameter. Die Funktion übergeht alle führenden Zeichen in `string`, bis eines gefunden wird, das nicht in `trennzeichen` enthalten ist. Dies ist der Anfang des ersten Teilstrings, der zurückgegeben wird.

Für weitere Aufrufe zum gleichen `string` übergibt man als ersten Parameter den Wert `NULL`. Der String mit Trennzeichen darf von Aufruf zu Aufruf wechseln. Die Suche wird jeweils hinter dem zuletzt gefundenen Trennzeichen fortgesetzt. Das jeweils gefundene Trennzeichen wird in `string` durch `\0` ersetzt und ein Zeiger auf den nächsten Teilstring geliefert, bis das Ende von `string` erreicht ist. Dann wird `NULL` zurückgegeben.

Achtung! Der ursprüngliche Inhalt von `string` wird durch das Überschreiben der Trennzeichen mit Nullen verändert! Deshalb darf man als ersten Parameter keine Stringkonstante übergeben.

→**strcspn()** (Seite 225), **strspn()** (Seite 229)

strtol()

```
#include <stdlib.h>

long int strtol( const char *string,
                 char **endezeiger,
                 int basis            );
```

verwandelt den String ab `string` in eine ganze Zahl, die zurückgegeben wird. Dabei werden aus `string` solange Zeichen gelesen, bis das Ende von `string` erreicht ist oder ein ungültiges Zeichen gelesen wird. Der dabei errechnete Zahlenwert wird zurückgegeben. Wenn `endezeiger` ungleich `NULL` ist, dann wird an der Stelle `*endezeiger` im Speicher ein Zeiger auf das erste Zeichen hinterlegt, das nicht mehr mit umgewandelt werden konnte. War bereits das erste Zeichen ungültig, dann ist der hinterlegte Zeiger der Wert von `string`.

Der String wird dabei als Zahl zur Basis `basis` betrachtet. Ist `basis` 0, dann darf die Ziffernfolge in `string` die Form einer *C*-Konstanten haben und die Basis wird aus den ersten Zeichen bestimmt (`0x` oder `0X` für Basis 16, `0` für Basis 8 etc.). Ansonsten darf `basis` Werte von 2 bis 36 annehmen, wobei die Buchstaben `a`–`z` bzw. `A`–`Z` für die Ziffern mit den Werten 10–35 stehen.

Kann kein Zeichen umgewandelt werden, dann wird 0 zurückgegeben. Ist der umzuwandelnde Wert betragsmäßig zu groß, dann wird `LONG_MIN` oder `LONG_MAX` zurückgegeben und `errno` auf `ERANGE` gesetzt.

→**atoi()** (Seite 180), **sscanf()** (Seite 224), **strtod()** (Seite 230),
→**strtoul()** (Seite 231)

strtoul()

```
#include <stdlib.h>

unsigned long int strtoul( const char *string,
                           char **endezeiger,
                           int basis            );
```

verwandelt den String ab `string` in eine ganze Zahl, die zurückgegeben wird. Dabei werden aus `string` solange Zeichen gelesen, bis das Ende von `string` erreicht ist oder ein ungültiges Zeichen gelesen wird. Der dabei errechnete Zahlenwert wird zurückgegeben. Wenn `endezeiger` ungleich `NULL` ist, dann wird an der Stelle `*endezeiger` im Speicher ein Zeiger auf das erste Zeichen hinterlegt, das nicht mehr mit umgewandelt werden konnte. War bereits das erste Zeichen ungültig, dann ist der hinterlegte Zeiger der Wert von `string`.

Der String wird dabei als Zahl zur Basis `basis` betrachtet. Ist `basis` 0, dann darf die Ziffernfolge in `string` die Form einer *C*-Konstanten haben und die Basis wird aus den ersten Zeichen bestimmt (`0x` oder `0X` für Basis 16, `0` für Basis 8 etc.). Ansonsten darf `basis` Werte von 2 bis 36 annehmen, wobei die Buchstaben `a`–`z` bzw. `A`–`Z` für die Ziffern mit den Werten 10–35 stehen.

Kann kein Zeichen umgewandelt werden, dann wird 0 zurückgegeben. Ist der umzuwandelnde Wert zu groß, dann wird `LONG_MAX` zurückgegeben und `errno` auf `ERANGE` gesetzt.

Außer der Behandlung des Ergebnisses als `unsigned` ist `strtoul()` identisch mit `strtol()`.

→**atoi()** (Seite 180), **sscanf()** (Seite 224), **strtod()** (Seite 230),
→**strtol()** (Seite 231)

system()

```
#include <stdlib.h>

int system( const char *kommando );
```

versucht den Kommandointerpreter der aufrufenden Programmumgebung zu rufen und das Kommando in `kommando` ausführen zu lassen.

Mit einem `NULL`-Zeiger als `kommando` kann man testen, ob ein Kommandoprozessor erreichbar ist. Dann wird nämlich ein von Null unterschiedlicher Wert zurückgegeben.

Der Inhalt des Strings `kommando` ist systemabhängig.

→**getenv()** (Seite 199)

tan()

```
#include <math.h>

double tan( double wert );
```

liefert den tan(`wert`) (`wert` ist im Bogenmaß anzugeben).

Ist `wert` zu groß, kann ein sehr ungenaues Ergebnis geliefert werden, ohne daß eine Fehlermeldung auftritt.

→**atan()** (Seite 178), **atan2()** (Seite 179), **tanh()** (Seite 232),
→**sin()** (Seite 222), **cos()** (Seite 186)

tanh()

```
#include <math.h>

double tanh( double wert );
```

liefert den tanh(`wert`).

→**exp()** (Seite 188), **sinh()** (Seite 223), **cosh()** (Seite 186)

time()

```
#include <time.h>

time_t time( time_t *zeit_zeiger );
```

liefert die aktuelle Systemzeit. Wenn `zeit_zeiger` ungleich `NULL` ist, dann wird die Zeit ebenfalls an dieser Stelle im Speicher hinterlegt.

Das folgende Beispiel gibt Datum und Uhrzeit in „üblicher“ Form aus:

```
/* futime.c 30. 7.95 kw
 */

#include <stdio.h>
#include <stddef.h>
#include <time.h>

void main()
{
  time_t      zeit;
  struct tm   z;
  char       *tag[] =
  {
    "Sonntag",
    "Montag",
    "Dienstag",
    "Mittwoch",
    "Donnerstag",
    "Freitag",
    "Samstag",
  };
  char       *mon[] =
  {
    "Januar",
    "Februar",
    "Maerz",
    "April",
    "Mai",
    "Juni",
    "Juli",
    "August",
    "September",
    "Oktober",
    "November",
    "Dezember",
  };

  /* Holen der komprimierten Uhrzeit in die Variable zeit:
   */
  zeit = time( NULL );

  /* Entpacken der Variablen zeit in die Struktur z:
   */
  z = *localtime( &zeit );

  /* Ausgabe von einigen Elementen der Struktur:                  */
  printf( "Heute ist %s, %d. %s %d  %02d:%02d h\n",
          tag[z.tm_wday],
          (int)z.tm_mday,
          mon[z.tm_mon],
          (int)z.tm_year+1900,
          (int)z.tm_hour,
          (int)z.tm_min
          );
```

```
}
```

→**ctime()** (Seite 187), **asctime()** (Seite 177), **gmtime()** (Seite 200),
→**localtime()** (Seite 204), **mktime()** (Seite 208)

tmpfile()

```
#include <stdio.h>

FILE *tmpfile( void );
```

öffnet eine temporäre Binärdatei, die nach dem Schließen automatisch entfernt wird (außer bei Programmabbruch durch `abort()`), im Modus `wb+` mit einem automatisch erzeugten Dateinamen. Geliefert wird der zugehörige Dateizeiger oder `NULL`, wenn keine Datei geöffnet werden kann.

→**fopen()** (Seite 192), **tmpnam()** (Seite 234), **exit()** (Seite 187), **abort()** (Seite 176)

tmpnam()

```
#include <stdio.h>

char *tmpnam( char *wohin );
```

erzeugt einen gültigen Dateinamen, der noch an keine Datei vergeben ist. Bei jedem Aufruf wird ein anderer Name erzeugt.

Wenn `wohin` den Wert `NULL` hat, dann wird der Name in einem statischen Speicherbereich erzeugt, der beim nächsten Aufruf überschrieben werden kann. Ansonsten wird der Name da erzeugt, worauf `wohin` weist. Dann sollte ab `wohin` Platz für mindestens `L_tmpnam` Zeichen sein.

Zurückgegeben wird ein Zeiger auf den erzeugten Namen, oder `NULL`, wenn ein Fehler auftrat.

→**tmpfile()** (Seite 234), **fopen()** (Seite 192), **remove()** (Seite 217)

tolower()

```
#include <ctype.h>

int tolower( int zeichen );
```

liefert zu einem Großbuchstaben (`A`–`Z`) den entsprechenden Kleinbuchstaben. Andere Zeichen werden nicht verwandelt.

Deutsche Umlaute werden nicht als Buchstaben behandelt, sondern als Sonderzeichen.

Achtung! Auf älteren Systemen (bei einigen Unixversionen) verwandelt `tolower()` alle Buchstaben; dann darf man die Funktion nur mit Großbuchstaben aufrufen.

→**toupper()** (Seite 235)

toupper()

```
#include <ctype.h>

int toupper( int zeichen );
```

liefert zu einem Kleinbuchstaben (`a–z`) den entsprechenden Großbuchstaben. Andere Zeichen werden nicht verwandelt.

Deutsche Umlaute werden nicht als Buchstaben behandelt, sondern als Sonderzeichen.

Achtung! Auf älteren Systemen (bei einigen Unixversionen) verwandelt `toupper()` alle Buchstaben; dann darf man die Funktion nur mit Kleinbuchstaben aufrufen.

→**tolower()** (Seite 234)

ungetc()

```
#include <stdio.h>

int ungetc( int zeichen, FILE *datei );
```

stellt das schon gelesen Zeichen `zeichen` wieder zurück in den Dateipuffer, so daß der nächste Leseaufruf wieder dieses Zeichen liefert (sofern zwischendurch keine Dateipositionierung erfolgt).

Man darf sich nur darauf verlassen, daß ein Zeichen zurückgestellt werden kann.

Zurückgegeben wird das Zeichen oder `EOF`, wenn mehrere Zeichen zurückgestellt werden sollen oder wenn `zeichen` den Wert `EOF` hat.

→**getc()** (Seite 199), **fgetc()** (Seite 189)

va_start(), va_arg(), va_end()

```
#include <stdarg.h>

void    va_start( va_list arglist, argument );
type??? va_arg  ( va_list arglist, type??? );
void    va_end  ( va_list arglist );
```

Mit diesen Makros kann man variable Parameterlisten (wie beispielsweise die Parameterliste für `printf()`) auswerten.

Eine Funktion, die mit einer variablen Parameterliste aufgerufen werden soll, vereinbart vom Typ `va_list` eine Variable. Hier soll sie `arglist` heißen, aber der Name ist natürlich frei wählbar.

Die Funktion braucht mindestens einen bekannten Parameter.

Der letzte bekannte Parameter (in der Deklaration vor der Zeichenfolge `...`) wird benutzt, um die Variable zu initialisieren:

```
va_start( arglist, letzter_bekannter_Parameter );
```

Dann kann man die variablen Parameter nacheinander mit je einem Aufruf von `va_arg()` holen, wobei der jeweilige Typ des zu holenden Parameters anzugeben ist. Soll beispielsweise eine `double` aus der Parameterliste in eine Variable `d` gelesen werden, dann ist

```
d = va_arg( arglist, double );
```

zu schreiben.

`va_arg()` hat keinen festgelegten Ergebnistyp; es liefert ein Objekt von dem Typ, den man als zweiten Parameter angibt.

Der zweite Parameter an `va_arg()` ist also kein Objekt (wie Konstante oder Variable), sondern eine Typenangabe!

So können alle Parameter geholt werden.

Vor Beendigung der Funktion muß unbedingt

```
va_end( arglist );
```

aufgerufen werden.

Beispiel:

```
/* fuvaarg.c 30. 7.95 kw
 */

#include <stdarg.h>
#include <stdlib.h>

/* maxabs() erhaelt einen Formatstring, der aus einer Folge
 * von Direktiven besteht. Jede Direktive beschreibt den Typ
 * eines weiteren Parameters aus der Parameterliste.
 * Die Direktiven sind im einzelnen:
 *    %d    (signed int)
 *    %u    (unsigned int)
 *    %ld   (signed long)
 *    %lu   (unsigned long)
 *    %f    (double)
 *    %s    (char *s, wobei s auf einen String zeigt, der
 *           wiederum eine Zahl enthaelt)
 * Zurueckgegeben wird der groesste Betrag aller Argumente.
 */

double maxabs( const char *s, ... )
{
  double      maximalwert = 0.0,
    wert;
  va_list     argp;
  char       *s2;

  if( s )
  {
    va_start( argp, s );    /* argp am letzten bekann-
                             * ten Argument ausrichten.
```

```
                          */
/* Schleife ueber Direktiven    */
while( *s )
{
  if( '%'==*s++ )     /* kommt hier '%'?       */
  {
    switch( *s++ )  /* Welche Direktive? */
    {
      /* Je nach dem Inhalt der
       * Direktive wird
       * ein Argument mit
       * dem jeweiligen Wert
       * aus der Parameterliste
       * geholt:
       */

      case 'd':
        wert = va_arg( argp, int );
        break;

      case 'u':
        wert = va_arg( argp, unsigned );
        break;

      case 'l':
        if( 'd'==*s )
          wert = va_arg( argp, long );
        else if( 'u'==*s )
          wert = va_arg( argp,
                         unsigned long
                         );
        s++;
        break;

      case 'f':
        wert = va_arg( argp, double );
        break;

      case 's':
        wert = strtod( va_arg( argp,
                               char * ),
                       NULL
                       );
        break;

      default:
        /* Ungueltige Direktiven uebergehen.
         */
        break;
    }

    /* Gegebenenfalls den bisherigen Maximal-
     * wert korrigieren:
     */
```

```
        if( wert>maximalwert )
          maximalwert = wert;
        else if( -wert>maximalwert )
          maximalwert = -wert;
      }
    }

    va_end( argp );         /* Auswertung beenden       */
  }  return maximalwert;
}

/* Das Hauptprogramm ruft maxabs() mit verschiedenen Parameter-
 * listen (hinsichtlich Anzahl und Datentypen) auf.
 */
int main()
{
  double w;

  w = maxabs( "%d%d", 123, 25 );
  printf( "w ist %f\n", w );

  w = maxabs( "%lu%s%d", 123ul, "25", -300 );
  printf( "w ist %f\n", w );

  w = maxabs( "%ld%s%d%f", 123ul, "250", -30, 3.141592E3 );
  printf( "w ist %f\n", w );

  return 0;
}
```

Die Ausgabe dieses Beispiels lautet:

```
w ist 123.000000
w ist 300.000000
w ist 3141.592000
```

→**vprintf()** (Seite 239), **vsprintf()** (Seite 238), **vfprintf()** (Seite 238)

vfprintf()

```
#include <stdio.h>

int vfprintf( FILE *datei, const char *format, va_list arglist );
```

ist identisch mit `fprintf()`, außer daß die variable Argumentliste nach `format` ersetzt wird durch einen Parameter vom Typ `va_list`.

→**va_start/va_arg/va_end()** (Seite 235), **fprintf()** (Seite 193)

vsprintf()

```
#include <stdio.h>

int vsprintf( char *s, const char *format, va_list arglist );
```

ist identisch mit `sprintf()`, außer daß die variable Argumentliste nach `format` ersetzt wird durch einen Parameter vom Typ `va_list`.

→**va_start/va_arg/va_end()** (Seite 235), **sprintf()** (Seite 223)

vprintf()

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

```
int vprintf( const char *format, va_list arglist );
```

ist identisch mit `printf()`, außer daß die variable Argumentliste nach `format` ersetzt wird durch einen Parameter vom Typ `va_list`.

→**va_start/va_arg/va_end()** (Seite 235), **printf()** (Seite 210)

A Tips am Rande

Hier werden in loser Reihenfolge interessante Möglichkeiten und Eigenheiten von *C* vorgestellt.

A.1 Vereinbarung von freien Feldgrenzen

In *C* ist es leider nicht möglich, direkt einen beliebigen Wert für die untere Grenze eines Feldes anzugeben; durch die enge Verknüpfung von Feldern einerseits und dem Zeiger auf das erste (bzw. nullte) Feldelement andererseits beginnen alle Felder mit dem Index 0. Dies ist für alte *FORTRAN*- oder *Pascal*-Programmierer lästig, da in deren Sprachen die Feldgrenzen beliebig angegeben werden dürfen.

In *C* beginnen nun alle Felder mit dem Index 0. Bei jedem Feldzugriff muß man den gewünschten Index um den (gedachten) untersten Index erniedrigen. Dadurch hat man zwei Wertebereiche für die Feldindices: zum einen den gewünschten Bereich aus der Problemstellung (logische Feldgrenzen), und zum anderen den physikalischen Wertebereich des vereinbarten Feldes (von 0 bis (*Anzahl der Elemente* - 1), also die realen Feldgrenzen).

Beispiel: Ein Programm soll von den Zahlen 10 bis 20 die Quadrate berechnen, in einem Feld speichern und danach alle ausgeben.

Der logische Indexbereich geht also von 10 bis 20; der reale hat die gleiche Anzahl Elemente (11), aber beginnt bei 0; er reicht also in *C* von 0 bis 10 (einschließlich).

In *Pascal* ist das Problem ebenso leicht:

```
PROGRAM Quadrate( OUTPUT );

VAR     i:        integer;
        feld:     ARRAY[10..20] OF integer;

BEGIN
    { Die Quadrate der Zahlen von 10 bis 20 berechnen:       }
    FOR i:=10 TO 20 DO
        feld[i]:=i*i;

    { Die berechneten Quadrate ausgeben:                     }
    FOR i:=10 TO 20 DO
        writeln( 'Das Quadrat von ', i, ' ist ', feld[i] );
END.
```

wie in *FORTRAN* zu lösen:

```
      PROGRAM QUADRA
      INTEGER     I, FELD(10:20)

*     DIE QUADRATE BERECHNEN:
      DO 100, I = 10, 20
        FELD(I) = I*I
 100  CONTINUE

*     UND AUSGEBEN:
      DO 200, I = 10, 20
```

```
        PRINT*, 'DAS QUADRAT VON ', I, ' IST ', FELD(I)
  200  CONTINUE

       END
```

Das Umrechnen des logischen Datenbereichs auf reale Adressen im Speicher übernimmt hier der Compiler.

In *C* dagegen muß man ein Feld mit dem untersten Index 0 vereinbaren und bei jedem Feldzugriff den gewünschten logischen Index (hier 10 bis 20) um die untere Feldgrenze (10) erniedrigen, um auf die realen Indices ab 0 zu kommen:

```
/* arrq0.c 30. 7.95 kw
 */

#include <stdio.h>
#include <math.h>

int main()
{
  long      feld[11];    /* Feld von 10 bis 20                */
  int       i;           /* Schleifenzaehler                  */

  /* Die Quadrate der Zahlen 10 bis 20 berechnen:
   */
  for( i=10; i<=20; i++ )
  {
    /* i laeuft von 10 bis 20, aber abgespeichert werden
     * muss in feld[0] bis feld[10]:
     */
    feld[i-10] = (long)i*(long)i;
  }

  /* Analog jetzt die zuvor berechneten Quadrate ausgeben:
   */
  for( i= 10; i<=20; i++ )
  {   printf( "Das Quadrat von %i ist %ld\n",
              i,
              feld[i-10]
              );
  }
}
```

Greift man an vielen Stellen im Programm auf das Feld zu, dann wird das Umrechnen der logischen Feldindices auf die realen (ab 0) unübersichtlich und fehlerträchtig. Durch Einführung eines zusätzlichen Zeigers kann man das Problem aber lindern. Dieser Zeiger soll den Typ „Zeiger auf ein Feldelement“ haben. In *C* darf man einen Zeiger ebenso wie einen Feldnamen mit einem Index versehen, um auf ein Element unter- oder oberhalb der entsprechenden Adresse zugreifen zu können. Der Zeiger wird nun so initialisiert, daß er nicht auf das niedrigste vereinbarte Feldelement zeigt, sondern um die untere Feldgrenze „daneben“. Ist die untere Feldgrenze positiv oder größer als die Feldlänge, dann weist der Zeiger neben das Feld. Das macht aber nichts, solange man als Index einen Wert von der unteren bis zur oberen Feldgrenze verwendet:

```
/* arrq1.c 30. 7.95 kw
 */

#include <stdio.h>
#include <math.h>

int main()
{
  long      feld[11];      /* Feld von 10 bis 20              */
  long
    *feldbasis =         /* Zeiger auf das (gedachte)       */
    &feld[-10];          /* nullte Element                  */
  int       i;             /* Schleifenzaehler                */

  /* Die Quadrate der Zahlen 10 bis 20 berechnen:
   */
  for( i=10; i<=20; i++ )
  {
    /* i laeuft von 10 bis 20, aber abgespeichert werden
     * muss in feld[0] bis feld[10].
     * Durch die Initialisierung sind aber feld[0] und
     * feldbasis[10] identisch; ebenso die jeweils
     * folgenden Elemente bis feld[10] und feldbasis[20]:
     */
    feldbasis[i] = (long)i*(long)i;
  }

  /* Analog jetzt die zuvor berechneten Quadrate ausgeben:    */
  for( i= 10; i<=20; i++ )
  {
    printf( "Das Quadrat von %i ist %ld\n",
            i,
            feldbasis[i]
            );
  }
}
```

Über den Zeiger `feldbasis` kann man endlich die gewünschten logischen Indices verwenden, die nicht bei 0 beginnen. Um ein versehentliches Manipulieren des Zeigers `feldbasis` zu vermeiden, ist er als `const` vereinbart.

Diese Vorgehensweise ist aber immer noch fehlerträchtig, da man bei jeder Feldvereinbarung von neuem den Vorgang genau durchblicken muß.

Durch den Praeprozessor von *C* kann man die Vereinbarung weitgehend automatisieren. Man definiert sich einmalig ein Makro namens `ARRAY` (am besten in einer einzigen `#include`-Datei) und vereinbart dann mit dessen Hilfe das eigentliche Feld und den Hilfszeiger. Den Namen des gewünschten Feldes verwendet das Makro für den Hilfszeiger; der Name des eigentlichen Feldes wird durch Anhängen von `_` vorweg und `_ARRAY_BASE` hinterher gebildet[40]. Die Bildung des Feldnamens erfolgt mit dem Praeprozessoroperator `##`.

Das Makro `ARRAY` erhält vier Parameter: Den Typ eines jeden Elementes, gewünschten Feldnamen, sowie die beiden Feldgrenzen.

[40] In der stillen Hoffnung, daß der resultierende Name nicht anderweitig vergeben wird.

```
/* arrq2.c 30. 7.95 kw
 */

#include <stdio.h>
#include <math.h>

#define ARRAY(typ,name,min,max)                                      \
                    typ  _ ## name ## _ARRAY_BASE[max-min+1];        \
                    typ *name = &_ ## name ## _ARRAY_BASE[-min]

int main()
{
  ARRAY(long,feld,10,20);         /* Feld von 10 bis 20       */
  int       i;                    /* Schleifenzaehler         */

  /* Die Quadrate der Zahlen 10 bis 20 berechnen:
   */
  for( i=10; i<=20; i++ )
  {
    /* Mit feld[10] bis feld[20] kann man jetzt die in
     * _feld_ARRAY_BASE[0] bis _feld_ARRAY_BASE[10]
     * vereinbarten 11 Feldelemente ansprechen:
     */
    feld[i] = (long)i*(long)i;
  }

  /* Analog jetzt die zuvor berechneten Quadrate ausgeben:
   */
  for( i= 10; i<=20; i++ )
  {
    printf( "Das Quadrat von %i ist %ld\n",
            i,
            feld[i]
            );
  }
}
```

Die Definition des Makros `ARRAY` kann man in einer Headerdatei hinterlegen und bei Bedarf einfach mit `#include` einfügen.

A.2 Potenzen mit ganzzahligen Exponenten

Mit der Funktion `pow()` aus der Standardbibliothek können Potenzen aus zwei Gleitkommazahlen gebildet werden.

Dies ist aber in vielen Fällen unnötiger Rechenaufwand, da Potenzen mit ganzzahligen Exponenten schneller berechnet werden können.

In [DEK II] und [Amm] sind recht schnelle Verfahren dazu angegeben.

Beispielhaft wird hierfür folgende Funktion angegeben:

```
/* popowi.c 30. 7.95 kw
 * Berechnung von Potenzen mit ganzzahligen Exponenten nach:
 *  Donald E. Knuth: The Art of Computer Programming
 *  Vol. II (Seminumerical Algorithms) Second Edition
 *  Abschnitt 4.6.3, S. 441, Algorithmus A.
 * Je nach Option kann man eine Funktion fuer eine ganzzahlige Basis
 * (#define TYP LONG) oder eine gebrochene Basis (#define TYP DOUBLE)
 * erzeugen.
 * Der Funktionsname ist dementsprechend lpowi() oder dpowi().
 *
 * Da in dem genannten Algorithmus weder n noch x nach Start der
 * Rechnung benoetigt werden, kann man auf die lokalen Variablen N und
 * Z (aus Knuth) verzichten. Daher folgende Variablenentsprechung:
 *      Knuth:      | powi.c:
 *
 *      n, N        | n
 *      Y           | y
 *      x, Z        | x
 *
 *
 * Zur leichteren Orientierung sind als Kommentar die entsprechenden
 * Fragmente des Originaltextes eingefuegt.
 */

/* LONG oder DOUBLE:
 */
#ifndef TYP
#define TYP         DOUBLE
#endif

#define LONG        1
#define DOUBLE      2

#if     TYP==DOUBLE
#define BASIS_T     double      /* Typ                           */
#define POW         dpowi       /* Funktionsname                 */
#elif   TYP==LONG
#define BASIS_T     long        /* Typ                           */
#define POW         lpowi       /* Funktionsname                 */
#else
error "TYP nicht LONG oder DOUBLE!"
#endif
```

```
/* TEST oder NTEST fuer oder gegen Testprogramm:
 */
#define TEST

BASIS_T POW( BASIS_T x, int n )
{
  BASIS_T     y = 1;          /* A1. [Initialize.]                */
  do
  {
    if( n & 0x0001 )            /* A2. [Halve N.] "wether N was
                                 *                even or odd"
                                 */
    {
      n >>= 1;            /* A2. [Halve N.] "N <- N/2"
                           */
      /* A3. [Multiply Y by Z.] "Set Y <- Z times Y"
       */
      y *= x;

      /* A4. [N = 0?] "If n=0, the algorithm
       *              terminates with Y as the answer."
       */
      if( !n ) return y;
    }
    else
    {
      n >>= 1;            /* A2. [Halve N.] "N <- N/2"
                           *                "if N was even,
                           *                skip to section
                           *                A5."
                           */
    }
    /* A5. [Square Z.]         "Set Z <- Z times Z"             */
    x *= x;
  }
  while( 1 );               /* "and return to step A2."         */
}

#ifdef TEST

#if     TYP==LONG
#define WRFMT           "%10ld"
#elif   TYP==DOUBLE
#define WRFMT           "%10.3f"
#else
error "TYP nicht LONG oder DOUBLE!"
#endif

#include <time.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

#define IMAX                10000000
```

```
void testpow( BASIS_T b, int exp )
{   BASIS_T     ret;
 double      dret;

 long        i;
 clock_t     start,
   ende;

 printf( "Testprogramm fuer Potenzieren mit "
         "ganzzahligen Exponenten.\n"
         );

 /* Test mit Originalroutine:                                    */
 start = clock();
 for( i=0; i<IMAX; i++ )
   dret = pow( (double)b, (double)exp );
 ende = clock();
 printf( "Original: pow(%10.3f,%7.1f) = %10.3f; in %.9f sec\n",
         (double)b,
         (double)exp,
         dret,
         (double)( ende - start )/((double)CLOCKS_PER_SEC*IMAX)
         );

 /* Test mit Algorithmus aus Knuth:                              */
 start = clock();
 for( i=0; i<IMAX; i++ )
   ret = POW( b, exp );
 ende = clock();
 printf( "Knuth:    pow(" WRFMT ",%7d) = " WRFMT "; in %.9f sec\n",
         b,
         exp,
         ret,
         (double)( ende - start )/((double)CLOCKS_PER_SEC*IMAX)
         );
 printf( "\n" );
}

int main()
{       testpow( 2, 3 );
 testpow( 2, 10 );
 return EXIT_SUCCESS;
}
#endif
```

In diesem Quelltext hat man mit `#define TYP DOUBLE` oder `#define TYP LONG` die Auswahl, eine von zwei verschiedenen Funktionen kompilieren zu lassen: Entweder die Funktion `long lpowi( long x, int n )` oder `double dpowi( double x, int n )`.

Weiterhin hat man mit `#define TEST` die Möglichkeit, ein Testprogramm erzeugen zu lassen. `#define NTEST` unterbindet dies; dann wird nur eine Funktion übersetzt.

Auf einem bestimmten Rechner (Laptop Pentium 3, 1.1GHz, Linux, `gcc 2.95.3`) werden von der `long`-Version folgende Ausgaben für 2^3 und 2^{10} erzeugt:

```
Testprogramm fuer Potenzieren mit ganzzahligen Exponenten.
Original: pow(     2.000,    3.0) =      8.000; in 0.000000170 sec
Knuth:    pow(         2,      3) =          8; in 0.000000011 sec

Testprogramm fuer Potenzieren mit ganzzahligen Exponenten.
Original: pow(     2.000,   10.0) =   1024.000; in 0.000000176 sec
Knuth:    pow(         2,     10) =       1024; in 0.000000018 sec
```

Dieser Algorithmus ist nicht so universell wie die Funktion `pow()` aus der Standardbibliothek, aber dafür in ihrem Einsatzgebiet wesentlich schneller.

Die Moral dieses Abschnitts ist damit, daß man die Rechenleistung von Programmen teilweise erheblich steigern kann, wenn man brauchbare Algorithmen für einen bestimmten Einsatzzweck entwickelt.

In der `double`-Version ist die Beschleunigung ebenfalls kräftig:

```
Testprogramm fuer Potenzieren mit ganzzahligen Exponenten.
Original: pow(     2.000,    3.0) =      8.000; in 0.000000160 sec
Knuth:    pow(     2.000,      3) =      8.000; in 0.000000015 sec

Testprogramm fuer Potenzieren mit ganzzahligen Exponenten.
Original: pow(     2.000,   10.0) =   1024.000; in 0.000000187 sec
Knuth:    pow(     2.000,     10) =   1024.000; in 0.000000022 sec
```

Auf anderen Rechnern kann der Vergleich aber ganz anders aussehen; solche Optimierungen sind in aller Regel nicht universell übertragbar.

In [Amm] sind teilweise noch etwas schnellere Algorithmen angegeben.

A.3 Empfehlungen zur Programmierung

Hier nur ein paar Stichpunkte, um die gröbsten Fehler zu vermeiden: Die Empfehlungen sind nur teilweise auf *C* begrenzt; aber durch die vielen Freiheiten und Fehlermöglichkeiten von *C* wird systematisches und diszipliniertes Vorgehen schnell unentbehrlich.

- *C* bietet neben der systemnahen Programmierung auch einen umfangreichen portablen Teil.

  Dies kann man nutzen, um möglichst viel der jeweiligen Problemlösung portabel zu schreiben.

  Wo das nicht machbar ist, sollte der systemspezifische Teil so klein wie irgend möglich sein. Weiterhin empfiehlt es sich, alle Systemabhängigkeiten deutlich als solche zu kennzeichnen.

- Den Praeprozessor ausnutzen!

  Mit `#define` kann man vor allem kleinere Programmänderungen schnell durchführen, wenn man die entsprechenden Konstanten rechtzeitig damit definiert.

  Mit `#include` sollte man dafür sorgen, daß alle Deklarationen und Definitionen, die mehrere Quelltexte betreffen, nur an einer Stelle stehen und dementsprechend auch nur an einer Stelle geändert werden müssen.

  Mittels bedingter Kompilierung und geschickter `#define`-Anweisungen kann man leicht verschiedene Versionen eines Programms verwalten: verschiedene Kunden, verschiedene Rechnerumgebungen, verschiedene Stufen der Ausgabemenge (ausführliche Protokollierung oder Fehlertests für die Entwicklung, schnellere Programmausführung für den Produktionsbetrieb), und so weiter.

  Siehe dazu auch `assert()` in Abschnitt 17.

- Den Quelltext übersichtlich gestalten: Den Inhalt aller Blöcke gegenüber dem umgebenden Block sauber einrücken.

  Beispiel:

```
#include <stdlib.h>

void vollelotte( char *zeiger, size_t wieviele )
{
  while( wieviele-- ) *zeiger++ = 0;
}

void pfuschrum( char **zeigerzeiger )
{
  if( (*zeigerzeiger=realloc( *zeigerzeiger, 800 )) )
  {
    vollelotte( *zeigerzeiger, 800 );
  }
  else printf( "\apfuschrum(): Speicher reicht nicht!\n" );
}

int main( int narg, char **args )
{
  char    *zeig;
  if( (zeig=malloc( (size_t)500 )) )
```

```
  {
    vollelotte( zeig, 500 );
    pfuschrum( &zeig );
  }
  else printf( "\amain(): Speicher reicht nicht!\n" );
}
```

- Ausgiebig und deutlich kommentieren! Auch Veränderungen nicht vergessen, sobald sie funktionieren, sondern zumindest im Quelltext als Kommentar beschreiben.

- Aussagefähige Namen verwenden; Namen nicht zu kurz wählen! Kurze Namen (`i`, `nmax`) sind in Ordnung, wenn sie nur in einem relativ kleinen Bereich verwendet werden (höchstens wenige Zeilen). Je länger jedoch der Gültigkeitsbereich ist, desto schwieriger ist es, den Sinn einer Variablen beim Lesen des Quelltextes zu erfassen; dann muß der Name entsprechend aussagekräftiger werden (`IndexSortiertesKundenfeld`, `MaximaleAnzahlKunden`). (Wer mit langen Namen zuviel Schreibarbeit hat, sollte nicht auf kurze Namen ausweichen, sondern auf einen vernünftigen Editor.)

  Dagegen Achtung bei Namen, die zwischen Quelltexten über den Linker ausgetauscht werden (externe Namen)! Diese sind auf vielen (älteren) Systemen nur in den ersten 6 oder 8 Zeichen signifikant.

  Möglichst unverfängliche und kollisionsfreie Namen verwenden.

  Namen nicht unnötig mit einem Tiefstrich beginnen lassen.

- Makronamen am besten schon durch die Schreibweise abheben. Es ist üblich, Makronamen durchweg groß zu schreiben (`MAX`) und alle anderen Namen entweder klein (`index`) oder gemischt (`ZeigeAenderung`).

- Die Namen von selbstdefinierten Datentypen sollte man auch von Objektnamen (Variablen, Funktionen) unterscheiden können. Meist hängt man an die Namen von Datentypen die Endung `_t` an.

- Viele *C*-Programmierer vergeben an Zeigervariablen nur Namen, die mit `p` beginnen (oder enden, je nach Vorliebe). Dementsprechend enthalten Namen von Zeigern auf Zeiger `pp` und so weiter.

- `goto` sehr sparsam verwenden!

  Die meisten Funktionen kommen gut ohne `goto` aus; manchmal ist eine Sprungadresse (*label*) sinnvoll, wenn an mehreren Stellen in einer Funktion ein ähnlicher Fehler auftreten kann (womöglich in tiefgeschachtelten `if`-Blöcken oder Schleifen), der mehrere Anweisungen zur Folge hat. Dann springt man im Fehlerfall zu einer Sprungadresse kurz vor Ende der Funktion und geht nur an dieser einen Stelle auf den Fehler ein.

  Achtung! Vor einer solchen Sprungadresse wird oft ein `return` vergessen, so daß die normale Programmabarbeitung auch in die Fehlerbehandlung eintritt.

  Braucht man irgendwann mehr als eine Sprungadresse innerhalb einer Funktion, dann sollte man ernsthaft über die Programmstruktur nachdenken.

  `longjmp()` ist im Programm noch wesentlich undurchsichtiger und unübersichtlicher; deshalb nur sehr überlegt einsetzen!

- Ein sehr häufiger Fehler ist die versehentliche Verwendung von `=` (anstatt `==`) zum Vergleich auf Gleichheit zweier Operanden. Dieser Fehler ist deshalb so tückisch, weil ein *C*-Compiler sich daran meistens gar nicht stört. Stattdessen wird eine Zuweisung kompiliert, wenn links vom Operator ein sogenannter *lvalue* steht (das ist ein Objekt, dem etwas zugewiesen werden darf, also beispielsweise eine Variable, die nicht als const vereinbart ist). Das Ergebnis der Zuweisung wird also anstelle des gewünschten Vergleichs auf *ungleich 0* getestet.

  Beispiel:

```
int antwort;
...
if( antwort='E' )      /* diese Zeile ist falsch! */
    tuwasmitE();
else
    wasanderes();
```

  Anstatt den Inhalt von `antwort` mit `'E'` zu vergleichen, wird an `antwort` der Wert `'E'` zugewiesen. Diese Zuweisung liefert immer ein Ergebnis ungleich 0 (nämlich `'E'`). Deshalb wird prinzipiell der Teil `tuwasmitE();` ausgeführt, unabhängig vom bisherigen Wert von `antwort`.

  In allen Fällen, in denen bei einem solchen Vergleich eine Konstante (hier: `'E'`), ein Rechenausdruck oder ein Funktionsaufruf steht, kann man den Fehler leicht vermeiden, indem man sich beizeiten angewöhnt, den betreffenden Operanden auf die linke Seite zu schreiben. Mit dieser Gewohnheit hätte man das zweite Gleichheitszeichen wahrscheinlich ebenfalls versehentlich wegfallen lassen, aber das Beispiel würde nun so aussehen:

```
int antwort;
...
if( 'E'=antwort )   /* diese Zeile ist auch falsch! */
    tuwasmitE();
else
    wasanderes();
```

  So bekommt man wenigstens eine Fehlermeldung vom Compiler, da an die Konstante nichts zugewiesen werden darf.

- Einer der beliebtesten Fehler ist die Verwendung ungültiger Zeiger: Entweder wurde der Zeiger nicht initialisiert oder er zeigt auf einen Speicherbereich, der inzwischen ungültig ist; siehe Seite 45.

- Ebenfalls häufig vergessen wird die `break`-Anweisung vor `case`- oder `default`-Anweisungen in einem `switch`-Block.

- Sehr gerne wird die obere Grenze eines Feldes überschritten: Bei einem Feld mit 10 Elementen hat das letzte Element den Index 9 (in Worten: neun)! Nicht zehn!

  Zur ungewohnten Zählweise (0 bis (n-1)) kommt bei Strings noch erschwerend hinzu, daß man für die abschließende 0 Platz vorsehen muß.

- *Den Quelltext beizeiten so gestalten, daß man als Programmierer unentbehrlich wird!*

  Wie geht das? Alles ausprobieren, wovon hier abgeraten wurde!

Literatur

[K&R I] Brian W. Kernighan, Dennis M. Ritchie:
The C Programming Language
1. Auflage
Prentice Hall
Veraltet, aber klassisch.
Die deutsche Übersetzung taugt nichts.

[K&R II] Brian W. Kernighan, Dennis M. Ritchie:
Programmieren in C
2. Ausgabe, ANSI C
Hanser Verlag, München/Wien 1990
relativ aktuell, leicht lesbar, mehr als Einführung in *C* denn zum Nachschlagen geeignet.

[C-Puzzle] Alan R. Feuer:
C-Puzzlebuch
C-Programmier Training
Hanser Verlag, München/Wien 1985
ISBN 3-446-14168-5
Witzige Denksportaufgaben, die teilweise sehr tief in *C* wühlen.

[c't port.] Dieter Brors:
Portable C-Funktionen
c't 1/1992, S. 221ff
Überblick über C-Funktionen, die einerseits über den ANSI-Standard hinausgehen , aber trotzdem auf vielen Systemen zu finden sind.

[c't C++] Roman Gerike:
Crash-Kurs in C++
Schnelleinstieg für Programmierer
c't 10/1991–1/1992
Überblick über die *C*-Weiterentwicklung *C++*.

[Stroustrup: C++] Bjarne Stroustrup:
Die C++ Programmiersprache
4., aktualisierte Auflage
Addison Wesley 2000
ISBN 3-8272-1660-X
Umfangreiche, vielleicht schwer verdauliche Einführung und Referenz; DAS Buch zu C++.

[DEK I] Donald E. Knuth:
The Art of Computer Programming
Second edition
Volume I: Fundamental Algorithms
Addison Wesley 1973
Grundlage für [DEK II] und [DEK III].

[DEK II] Donald E. Knuth:
The Art of Computer Programming
Second edition
Volume II: Seminumerical Algorithms
Addison Wesley 1973
Grundlagen von Rechnerarithmetik, Theorie zu elementaren Rechenverfahren.

[DEK III] Donald E. Knuth:
The Art of Computer Programming
Second edition
Volume III: Sorting and Searching
Grundlagen von Sortier- und Suchalgorithmen, sehr ausführlich, theoretisch und trocken, nicht für „eben mal nachschlagen“, ignoriert *C* ebenso wie andere konkrete Sprachen.

[EM] G. Engeln-Müllges/F. Reutter:
Formelsammlung zur numerischen Mathematik mit C-Programmen
Bibliographisches Institut, Mannheim 1986
Enthält Unterprogrammsammlung mit breitem Spektrum; Algorithmen in *C* auf Diskette erhältlich (analog: *FORTRAN/Pascal/Basic/Modula2*.)

[Amm] Leendert Ammeraal:
Programmdesign und Algorithmen in C
Praxisorientierte Anleitung für Fortgeschrittene.
ISBN 3-446-15864-2
Einige interessante Algorithmen, altes *K&R-C*;
zum Thema *Programmdesign* weniger interessant.

[DudInf] *Duden Informatik*
Ein Sachlexikon für Studium und Praxis
hrsg. Lektorat BI-Wiss.-Verlag
Dudenverlag Mannheim, Wien, Zürich 1988
ISBN 3-411-02421-6

[Tan] Andrew S. Tanenbaum:
Operating Systems
Design and Implementation
Prentice Hall 1987
ISBN 0-13-637331-3 025
Interessantes zu Unix-ähnlichen Betriebssystemen (MINIX) und Systemprogrammierung allgemein. Alles in englisch und *K&R-C*. Teilweise witzig zu lesen.

[UmstANSI] UNIX System Laboratories, Inc.
UNIX SYSTEM V RELEASE 4
Leitfaden für die Umstellung auf ANSI-C
Carl Hanser Verlag, Münster 1991
ISBN 3-446-16255-0
Saubere Erklärung der Unterschiede zwischen altem K&R-C und neuem ANSI-C, variable Parameterlisten, arithmetische Umwandlungen, Präprozessor, Mehrbytezeichen für Asien, u. m.

[c’t Rek.] Heinz Ebert:
Weltanschauliches in Schachteln
Vom Sinn und Unsinn der Rekursion
c’t 10/1986, S. 78ff
Lesenswert zum Thema Rekursion, Beispiele in *Pascal.*

[c’t Sort] Dieter Porth:
Sortierverfahren
Die klassischen Algorithmen im Überblick

c't 3/1992, S. 263ff
Kurzabriß zum Nachschlagen mit *Pascal*-Beispielen.

[c't Opt. 1] Leo Drisis:
Hohes C
C-Programme einfach optimiert c't 6/1990, S. 334ff
Tips zur Optimierung auf Quelltextebene, Beispiele in *C*.

[c't Opt. 2] Andreas Dieberger:
Optimierung an der Quelle
c't 1/1991–4/1991
Tips zur Optimierung auf Quelltextebene, Beispiele in *Pascal*.

[c't Zahlen] Gerhard Mühlsteph:
Wie Computer rechnen
Zahlendarstellung und Zahlenverarbeitung
c't 3/1987, S. 203ff
Einblick in die binäre Zahlendarstellung.

[c't make] Klaus Zerbe:
Make ohne Macken
Vom Umgang mit dem populärsten Projektverwaltungsprogramm
c't 11/1991, S. 202ff
Benutzung des Hilfsprogramms *make*.

[KW: EinfInf] Klaus Wachtler:
Skript zur Vorlesung Einführung in die Informatik
Selbstverlag 2001, nichtkommerzielle Nutzung frei
email: AnyWare@Wachtler.de
Postscript: `http://www.wachtler.de/skript_einfinf.ps.gz`
HTML: `http://www.wachtler.de/skript_einfinf.tar.gz`
Online: `http://www.wachtler.de/skript_einfinf/`
Grundlagen: Digitalrechner, Programme, Prozesse, Zahlensysteme; noch unvollständig!

B Tabellen

Zum schnelleren Nachschlagen sind hier nochmals die am häufigsten benötigten Tabellen aufgeführt.

B.1 Die Operatoren in C

Eine genauere Beschreibung steht in **11 Operatoren und Ausdrücke** auf Seite 83.

<table>
<tr><th>Operator</th><th>Zusammenfassung</th><th>Bedeutung</th></tr>
<tr><td><code>() []</code>
<code>-> .</code></td><td>von links</td><td>Klammeroperator, Feldzugriff
Elementauswahl für <code>struct/union</code></td></tr>
<tr><td><code>! ~</code>
<code>++ -- + -</code>
<code>* &</code>
<code>(...) sizeof</code></td><td>von rechts</td><td>log. und bin. Verneinung
Inkrement, Dekrement, Vorzeichen
Zeigerverweis, Adreßoperator
cast, Speichergröße</td></tr>
<tr><td><code>* / %</code></td><td>von links</td><td>Multiplikation, Division, modulo</td></tr>
<tr><td><code>+ -</code></td><td>von links</td><td>Addition, Subtraktion</td></tr>
<tr><td><code><< >></code></td><td>von links</td><td>links, rechts Schieben</td></tr>
<tr><td><code>< <= > >=</code></td><td>von links</td><td>numerischer Vergleich</td></tr>
<tr><td><code>== !=</code></td><td>von links</td><td>Gleichheit, Ungleichheit</td></tr>
<tr><td><code>&</code></td><td>von links</td><td>binäres Und (AND)</td></tr>
<tr><td><code>^</code></td><td>von links</td><td>binäres Exklusiv-Oder (XOR)</td></tr>
<tr><td><code>|</code></td><td>von links</td><td>binäres Oder (OR)</td></tr>
<tr><td><code>&&</code></td><td>von links</td><td>logisches Und (AND)</td></tr>
<tr><td><code>||</code></td><td>von links</td><td>logisches Oder (OR)</td></tr>
<tr><td><code>?:</code></td><td>von rechts</td><td>Auswahloperator</td></tr>
<tr><td><code>= += -= *= /=</code>
<code>%= &= ^= |=</code>
<code><<= >>=</code></td><td>von rechts</td><td>Zuweisungen</td></tr>
<tr><td><code>,</code></td><td>von links</td><td>Kommaoperator</td></tr>
</table>

B.2 Ersatzzeichenfolgen für `char`- und Stringkonstanten

Zeichen	*Schreibweise*
Zeilenvorschub, Zeilentrenner	`'\n'`
Alarmzeichen	`'\a'`
Tabulatorzeichen	`'\t'`
Vertikaltabulator	`'\v'`
backspace	`'\b'`
carriage return	`'\r'`
Seitenvorschub (*form feed*)	`'\f'`
linker Schrägstrich (*back slash*)	`'\\'`
Fragezeichen	`'\?'`
Apostroph	`'\''`
Gänsefüßchen	`'\"'`

B.3 Formatstring für `printf()`, `fprintf`, etc.

Der Formatstring besteht aus auszugebenden Zeichen und Steuersequenzen.

Jede Steuersequenz beginnt mit einem Prozentzeichen (`%`) und endet mit der eigentlichen Direktive (`d`, `i`, `o`, etc.):

Direktive:	*Argument:*	*Ausgabe:*	*Besonderheiten:*
`%d`	`int`	Dezimalzahl	*Genauigkeit*= Mindestanzahl Zeichen
`%i`	`int`	Dezimalzahl	*Genauigkeit*= Mindestanzahl Zeichen
`%u`	`unsigned int`	Dezimalzahl	*Genauigkeit*= Mindestanzahl Zeichen
`%o`	`unsigned int`	Oktalzahl	*Genauigkeit*= Mindestanzahl Zeichen `#` erzeugt eine führende Null.
`%x`	`unsigned int`	Sedezimalzahl (Basis 16)	Ziffern `0`–`9`, `a`–`f`
`%X`	`unsigned int`	Sedezimalzahl (Basis 16)	Ziffern `0`–`9`, `A`–`F`
`%e`	`double`	zum Beispiel `1.234567e-15`	*Genauigkeit*= Anzahl Nachkommastellen `#` gibt Dezimalpunkt auch aus, wenn *Genauigkeit*`==0` ist.
`%E`	`double`	zum Beispiel `1.234567E-15`	*Genauigkeit*= Anzahl Nachkommastellen `#` gibt Dezimalpunkt auch aus, wenn *Genauigkeit*`==0` ist.
`%f`	`double`	zum Beispiel `1.234567`	*Genauigkeit*= Anzahl Nachkommastellen
`%g`	`double`	wie Format `%e` oder `%f`, je nach Zahlenwert	*Genauigkeit*= Anzahl Nachkommastellen `#` verhindert das Entfernen von abschließenden Nullen
`%G`	`double`	wie Format `%E` oder `%f`, je nach Zahlenwert	`#` verhindert das Entfernen von abschließenden Nullen
`%c`	`int`	ein Zeichen des Zeichensatzes	
`%s`	`char *`	Inhalt einer nullterminierten Zeichenkette	
`%p`	`void *`	Wert des Zeigers (also eine Adresse)	`#` bewirkt Voranstellen von `0x`
`%n`	`int *`	—	An der Stelle im Speicher, auf die das Argument zeigt, wird die Anzahl der bisher mit diesem Funktionsaufruf ausgegebenen Zeichen abgelegt.
`%%`	—	Prozentzeichen `%`	

Zwischen dem einleitenden Prozentzeichen und der Direktive darf eine Auswahl folgender Steuerbefehle eingesetzt werden:

Steuerbefehl:	*Wirkung:*
`+`	positive Zahlen werden mit Vorzeichen ausgegeben
`␣` (*Leerz.*)	positive Zahlen werden mit führendem Leerzeichen ausgegeben
`0`	Es wird auf die Feldbreite nicht mit Leerzeichen, sondern mit Nullen aufgefüllt
`-`	Ausgabe linksbündig in der Feldbreite
`#`	alternative Ausgabe, je nach Direktive
Feldbreite (als Zahl) (oder als `*`)	Mindestzahl der auszugebenden Zeichen (ggf. Auffüllen mit Leerzeichen) Mindestzahl muß noch vor dem auszugebenden Argument als `short int` übergeben werden
Genauigkeit (als Punkt `.` mit nachfolgender Zahl) (oder als `.*`)	auszugebende Genauigkeit (max. Länge bei `%s`) Genauigkeit muß noch vor dem auszugebenden Argument als `short int` übergeben werden

Objektgröße: einer der Buchstaben:
`h`,
`l`,
`L`
	Speichergröße des auszugebenden Objekts:
	
	`short int` statt `int`
	`long int` statt `int`
	`long double` statt `double`

B.4 Formatstring für `scanf()`, `fscanf`, etc.

Der Formatstring besteht aus einzulesenden Zeichen (!) und Steuersequenzen. In aller Regel will man die zu einzulesenden Zeichen nicht vorgeben, also besteht der Formatstring meist nur aus Steuersequenzen. Weiterhin kann im Formatstring *white space* vorkommen; (falls vorhanden) wird an dieser Stelle jeglicher *white space* im Eingabestrom überlesen und ignoriert.

Die Funktion `scanf()` und Ihre Anverwandten liefern als Rückgabewert die Anzahl der erfolgreich bearbeiteten Direktiven. Dabei wird `%n` aber nicht mitgezählt.

Jede Steuersequenz beginnt mit einem Prozentzeichen (`%`) und endet mit der eigentlichen Direktive (`d`, `i`, `o`, etc.):

Direktive:	*Argument:*	*Eingabe:*	*Besonderheiten:*
`%d`	`int *`	ganze Dezimalzahl	
`%i`	`(unsigned) int *`	ganze Zahl wie C-Konstanten, z.B. `27` (dezimal), `0x1b` (sedezimal) oder `033` (oktal)	Zahlenbasis ergibt sich aus dem Anfang der Zahl
`%u`	`unsigned int *`	nichtnegative Dezimalzahl	
`%o`	`unsigned int *`	nichtnegative Oktalzahl	
`%x`, `%X`	`unsigned int *`	nichtnegative Sedezimalzahl	
`%e`, `%E`, `%f`, `%F`, `%g`, `%G`	`float *`	Gleitkommazahl mit/ohne Vorzeichen, mit/ohne Exponent, z.B. `-12.3E-8`	
`%c`	`char *`	Zeichen	*white space* wird nicht überlesen, sondern verwendet. 1 oder *Feldbreite* Zeichen werden gelesen. Es wird **keine** abschließende `\0` angehängt.
`%s`	`char *`	Zeichenkette ohne *white space*	Es wird eine abschließende `\0` angehängt (gegebenenfalls zusätzlich zu einer angegebenen Feldbreite).
...			

... Direktive:	Argument:	Eingabe:	Besonderheiten:
`%[Zeichenfolge]`	`char *`	alle Zeichen in *Zeichenfolge*	*white space* wird nicht überlesen, sondern verwendet. Wenn `]` in *Zeichenfolge* vorkommen soll, dann als erstes Zeichen, z.B. `%[]ab]`
`%[^Zeichenfolge]`	`char *`	alle Zeichen, die nicht in *Zeichenfolge* vorkommen	*white space* wird nicht überlesen, sondern verwendet. Wenn `]` in *Zeichenfolge* vorkommen soll, dann als erstes Zeichen, z.B. `%[^]ab]`
`%p`	`void **`	sedezimale Adresse	Eingabe wird nach `void *` konvertiert.
`%n`	`unsigned int *`	—	Die Anzahl bisher von diesem Formatstring eingelesenen Zeichen wird an der Stelle im Speicher hinterlegt, auf die das Argument zeigt (zählz nicht zum Rückgabewert).
`%%`	—	Prozentzeichen `%`	*white space* wird nicht überlesen.

Zwischen dem einleitenden Prozentzeichen und der Direktive darf eine Auswahl folgender Steuerbefehle eingesetzt werden (außer bei `%%`):

Steuerbefehl:	*Wirkung:*
`*`	Die Eingabe, die von dieser Direktive abgedeckt wird, soll überlesen und nicht gespeichert werden. Dementsprechend darf zu dieser Direktive auch kein Argument (siehe obige Tabelle) angegeben werden.
Feldbreite	Anzahl der maximal zu lesenden Zeichen der Eingabe (positive ganze Zahl)
Typ	Ein Buchstabe, der den Typ des erwarteten Arguments verändert. (siehe oben): `h` erwartet `short int *` statt `int *`. `l` erwartet `long int *` statt `int *`, bzw. `double *` statt `float *`. `L` erwartet `long double *` statt `float *`

B.5 ASCII-Tabelle

Nicht alle, aber doch die meisten Rechner verwenden den ASCII-Zeichensatz. Deshalb hier die Tabelle mit den Zeichen 0 bis 127.

Zuerst die untere Hälfte, auf der nächsten Seite folgt die zweite Hälfte:

Dez.	*Hex.*	*Zeichen*	*C-Darst.*	*Dez.*	*Hex.*	*Zeichen*	*C-Darst.*
0	0x00	NUL	'\0'	32	0x20	space	' '
1	0x01	SOH		33	0x21	!	'!'
2	0x02	STX		34	0x22	"	'\"'
3	0x03	ETX		35	0x23	#	'#'
4	0x04	EOT		36	0x24	$	'$'
5	0x05	ENQ		37	0x25	%	'%'
6	0x06	ACK		38	0x26	&	'&'
7	0x07	BEL	'\a'	39	0x27	´	'´'
8	0x08	BS	'\b'	40	0x28	(	'('
9	0x09	HT	'\t'	41	0x29	)	')'
10	0x0A	LF	'\n'	42	0x2A	*	'*'
11	0x0B	VT	'\v'	43	0x2B	+	'+'
12	0x0C	FF	'\f'	44	0x2C	'	'\''
13	0x0D	CR	'\r'	45	0x2D	-	'-'
14	0x0E	SO		46	0x2E	.	'.'
15	0x0F	SI		47	0x2F	/	'/'
16	0x10	DLE		48	0x30	0	'0'
17	0x11	DC1		49	0x31	1	'1'
18	0x12	DC2		50	0x32	2	'2'
19	0x13	DC3		51	0x33	3	'3'
20	0x14	DC4		52	0x34	4	'4'
21	0x15	NAK		53	0x35	5	'5'
22	0x16	SYN		54	0x36	6	'6'
23	0x17	ETB		55	0x37	7	'7'
24	0x18	CAN		56	0x38	8	'8'
25	0x19	EM		57	0x39	9	'9'
26	0x1A	SUB		58	0x3A	:	':'
27	0x1B	ESC		59	0x3B	;	';'
28	0x1C	FS		60	0x3C	<	'<'
29	0x1D	GS		61	0x3D	=	'='
30	0x1E	RS		62	0x3E	>	'>'
31	0x1F	US		63	0x3F	?	'\?'

Dez.	*Hex.*	*Zeichen*	*C-Darst.*	*Dez.*	*Hex.*	*Zeichen*	*C-Darst.*
64	0x40	@	'@'	96	0x60	`	'`'
65	0x41	A	'A'	97	0x61	a	'a'
66	0x42	B	'B'	98	0x62	b	'b'
67	0x43	C	'C'	99	0x63	c	'c'
68	0x44	D	'D'	100	0x64	d	'd'
69	0x45	E	'E'	101	0x65	e	'e'
70	0x46	F	'F'	102	0x66	f	'f'
71	0x47	G	'G'	103	0x67	g	'g'
72	0x48	H	'H'	104	0x68	h	'h'
73	0x49	I	'I'	105	0x69	i	'i'
74	0x4A	J	'J'	106	0x6A	j	'j'
75	0x4B	K	'K'	107	0x6B	k	'k'
76	0x4C	L	'L'	108	0x6C	l	'l'
77	0x4D	M	'M'	109	0x6D	m	'm'
78	0x4E	N	'N'	110	0x6E	n	'n'
79	0x4F	O	'O'	111	0x6F	o	'o'
80	0x50	P	'P'	112	0x70	p	'p'
81	0x51	Q	'Q'	113	0x71	q	'q'
82	0x52	R	'R'	114	0x72	r	'r'
83	0x53	S	'S'	115	0x73	s	's'
84	0x54	T	'T'	116	0x74	t	't'
85	0x55	U	'U'	117	0x75	u	'u'
86	0x56	V	'V'	118	0x76	v	'v'
87	0x57	W	'W'	119	0x77	w	'w'
88	0x58	X	'X'	120	0x78	x	'x'
89	0x59	Y	'Y'	121	0x79	y	'y'
90	0x5A	Z	'Z'	122	0x7A	z	'z'
91	0x5B	[	'['	123	0x7B	{	'{'
92	0x5C	\	'\\'	124	0x7C	|	'|'
93	0x5D	]	']'	125	0x7D	}	'}'
94	0x5E	^	'^'	126	0x7E	~	'~'
95	0x5F	_	'_'	127	0x7F	DEL	

Index