INFORMATIK II

While-Programme, Datenstrukturen
und Funktionen (C-Version)

Fachhochschule Fulda
Fachbereich Elektrotechnik
Prof. Dr. Timm Grams

Datei: inform_2c.doc
14. Februar 2011
(Erstausgabe: 11. Februar 2000)

Hinweise

Konzept: Das Auswendiglernen einer bestimmten Programmiersprache kann nicht Kern der Informatik-Ausbildung sein. Es ist es heute noch weniger als vor ein, zwei Jahrzehnten. Gründe:

1. Der Ingenieur wird in der Praxis mehreren Sprachen begegnen. Heute C, morgen FORTRAN, und Java, Object Pascal, Visual C, Smalltalk, Delphi, Eiffel, ...
2. Heute sind die Sprachen so umfangreich, dass man sie nicht komplett im Kopf behalten kann[1].
3. Wegen der heute sehr umfangreichen und leistungsstarken Bibliotheken gibt es einen *Trend vom Schreiben zum Lesen:* Der angehende Software-Entwickler muss flink sein beim Durchstöbern von Bibliotheken. Dazu braucht er Orientierungshilfen.

Der richtige Zugang zur Programmierung und zu den Sprachen wird wichtiger. Der Königsweg führt über die Grammatik einer Sprache. Da setzt diese Informatik-Einführung an: Darstellung der Syntax einer Sprache, Vorstellung der Semantik wesentlicher Sprachkonstrukte und die Einübung des Umgangs mit der Sprache anhand typischer Beispiele. Ansonsten wird auf die Quellen verwiesen; dort findet man die Details und die Definitionen.

Literaturangaben in Kurzform beziehen sich auf die Literaturverzeichnisse in Teil I und Teil II. Hinsichtlich der Spracheigenschaften von C wird grundsätzlich auf das Werk von Brian W. Kernighan und Dennis M. Ritchie als Standardreferenz zurückgegriffen. Das wird in den Literaturhinweisen nicht immer wieder erwähnt.

Zur Typographie: Programme und insbesondere deren Variablen werden grundsätzlich nicht kursiv geschrieben. Kursiv stehen alle Variablen und Funktionsbezeichner, die nicht Bezeichner in einem Programm sind. Umfangreichere Programmteile und Programme in separaten Absätzen werden in Schreibmaschinenschrift (`Courier`) dargestellt.

Begleitmaterial: Das Skriptum ist gedacht als Leitfaden der Lehrveranstaltung; es ist *nicht* gedacht als Lehrbuch der Programmiersprache `C`. Ausgezeichnete Lehbücher darüber gibt es für jeden Geschmack und in reicher Auswahl. Für eine erste Übersicht empfiehlt sich eine *Online-Suche* im Katalog der FH-Bibliothek. Eine von mir empfohlene Auswahl enthält das folgende Literaturverzeichnis. Zu einigen Themen sind auf meiner Informatik-Web-Seite Animationen (PowerPoint-Foliensätze) zu finden. Adresse:

`http://www.fh-fulda.de/~grams/informat.htm`

Zur Programmiersprache: Die dem Einführungskurs in die Informatik zugrundeglegte Programmiersprache C eignet sich nicht besonders gut als Schulsprache[2]. Die Wahl fällt auf die

---

[1] Die Idealvorstellung, dass die Sprache so einfach sein sollte, "dass ein Programmierer sie vollständig lernen und sich alle ihre Eigenschaften ins Gedächtnis rufen kann" (C. A. R. Hoare), wird heute bestenfalls noch von Schulsprachen wie Oberon erfüllt - also von Sprachen, die in der Praxis kaum Bedeutung haben. Schulsprachen können aktuelle Konzepte ohne historischen Ballast verwirklichen. Bei Praxissprachen geht das naturgemäß nicht.

[2] Das hier angesprochene Pro und Kontra wurde in Informatikkreisen ausgiebig diskutiert. Zu den Hauptmängeln von C gehören die nicht immer eindeutige Datentypen (boolesche Konstante beispielsweise werden durch Zahlen dargestellt), die mit verschiedenen Bedeutungen überladenen Symbole (insbesondere die runden Klammern und das Komma), und die sprachtypische Nutzung von Nebenwirkungen in Ausdrücken. Es kann leicht zu Programmierfehlern kommen, und die Programmierwerkzeuge können nur wenig zu ihrer Aufdeckung beitragen. In C ist es möglich, äußerst knappe und redundanzarme Programme zu schreiben. Hat der Programmierer einen Fehler gemacht, ist dem Programm dann oft die ursprüngliche Absicht nicht mehr anzusehen. Programmierwerkzeuge sind mit der Fehlerdiagnose überfordert und sie liefern dann eher verwirrende Hinweise.

Sprache eher deshalb, weil sie in der Praxis weit verbreitet ist. Aus diesem Grunde sind einige Kompromisse einzugehen: Die Sprache wird nicht im vollen Umfang vorgestellt. In der Einführung werden nur die wichtigsten und gut beherrschbaren Elemente der Sprache verwendet. Wer die Sprache voll ausnutzen will, sollte auf die ausgezeichnete Beschreibung der Sprache durch ihre Erfinder Kernighan und Ritchie (1988) zurückgreifen.

Der hier propagierte Programmierstil – die sogenannte *strukturierte Programmierung* - kommt ohne die *Goto*-Anweisung und ohne *Marken* (Labels) aus. Diese Dinge werden folglich in dieser Einführung vorsätzlich weggelassen.

Zur Übung: Am Schluss einer jeden Lektion sind Übungsaufgaben gestellt. Diese Übungen sind im *Selbststudium* zu bearbeiten. In den Übungsstunden der folgenden Woche werden ausschließlich Fragen behandelt, die sich im Laufe dieses Selbststudiums ergeben haben. Diese Fragen sind auf speziellen Blättern, die zu Beginn der Stunde verteilt werden, zu notieren. Bei Fragen von allgemeinem Interesse geschieht deren Beantwortung im Rahmen gesamten Gruppen, ansonsten gibt es individuelle Beratung. Vom Lehrenden werden grundsätzlich *keine Musterlösungen* vorgetragen. In den Übungsstunden gehen sämtliche *Aktivitäten von den Studierenden* aus.

Zum Praktikum: Für das Praktikum werden im Hochschulnetz eigene Unterlagen bereitgestellt. Dort ist auch alles Organisatorische zum Praktikum zu finden.

Die objektorientierte Erweiterung C++ macht die Sache noch schlimmer. Der Diskussionsbeitrag von K. Hug zum Artikel "C++ im Nebenfachstudium: Konzepte und Erfahrungen" im Informatik-Spektrum 19 (1996) 6, 338-341, entspricht auch meinen Erfahrungen.

Seit einigen Jahren exisitiert mit der Programmiersprache Java eine – in wesentlichen Teilen ebenfalls auf C basierende - gute Alternative zu C++. Es ist deshalb vertretbar, die Programmierausbildung mit einer Untermenge von C zu beginnen und dies als Einstieg in die objektorientierte Programmierung mit Java zu nutzen.

Gliederung

Teil II

Literatur

Programmiersprachen: C, C++, Java

Aho, A. V.; Sethi, R.; Ullman, J. D.: Compilers. Principles, Techniques, and Tools. Addison-Wesley, Reading, Mass. 1986. *Das berühmte „Drachen-Buch“ von Aho, Sethi und Ullman stellt Prinzipien dar, nach denen Rechner Programme in Maschinenform überführen und interpretieren; und es zeigt deren Anwendung auf die heute gängigen Programmiersprachen wie* `Fortran`, `Pascal` *und* C. *Hier werden diese Prinzipien benutzt, um einfache mentale Modelle des Rechnens zu entwickeln (Modellcomputer, „Papiercomputer“). Das soll helfen, Programmierfehler zu vermeiden.*

Darnell, P. A.; Margolis, P. E.: C, a software engineering approach. Springer, New York 1991. *Enthält eine Beschreibung der Standardbibliothek (ANSI Runtime Library).*

Goll, J.; Grüner, U.; Wiese, H.: C als erste Programmiersprache. ISO-Standard. Teubner, Leipzig 1999. *Eines der vielen Bücher, die dem Anfänger beim Einstieg in C helfen wollen. Eröffnet eine etwas andere Sicht auf die in der Lehrveranstaltung behandelten Dinge.*

Groß, S.: Skriptum zur Lehrveranstaltung „Sonderprobleme der Informatik: Programmieren in C“ des Fachbereichs Angewandte Informatik, Fulda, 1998. *Die Darstellung der lexikalischen Elementen und der Standardbibliothek habe ich teilweise daraus übernommen.*

Kernighan, B. W.; Ritchie, D. M.: The C Programming Language. 2nd edition. Prentice Hall, Englewood Cliffs, New Jersey 1988. *Standardwerk zu C von den Schöpfern der Sprache. Bevorzugte Referenz.*

Stroustrup, B.: The C++ Programming Language. 2nd edition. Addision-Wesley, Reading, Massachusetts 1995. *Quelle von Beispielen.*

Wirth, N.: Compilerbau. Teubner, Stuttgart 1986

Programmiermethodik

Aho, A. V.; Hopcroft, J. E.; Ullman, J. D.: Data Structures and Algorithms. Addison-Wesley, Reading, Massachusetts 1983

Alagić, S.; Arbib, M. A.: The Design of Well-Structured and Correct Programs. Springer-Verlag, New York 1978

Baber, R. L.: Fehlerfreie Programmierung für den Software-Zauberlehrling. Oldenbourg, München, Wien 1990

Balzert, H.: Lehrbuch Grundlagen der Informatik. Konzepte und Notationen in UML, Java und C++, Algorithmik und Software-Technik, Anwendungen. Spektrum Akademischer Verlag, Heidelberg 1999

Dahl, O.-J.; Dijkstra, E. W.; Hoare, C. A. R.: Structured Programming. Academic Press, London 1972. *Ein Standardwerk der Programmierung von den Pionieren der Zunft.*

Gries, D.: The Science of Programming. Springer Heidelberg 1981

Gumm, H.-P.; Sommer, M.: Einführung in die Informatik. Oldenbourg, München 1998. *Umfassende Darstellung der Grundlagen der heutigen Informatik.*

Knuth, D.: The Art of Computer Programming. Vol. 1: Fundamental Algorithms. Addison-Wesley 1973. *Knuths Bücher sind eine Fundgrube für den Programmierer. Hier findet er grundlegende Strukturen, Verfahren und mathematischen Hintergrund.*

Knuth, D.: The Art of Computer Programming. Vol. 2: Seminumerical Algorithms. Addison-Wesley 1981

Linger, R. C.; Mills, H. D.; Witt, B. I.: Structured Programming. Theory and Practice. Addison-Wesley, Reading, Mass. 1979. *Die Semantik von Programmen wird in einer Weise eingeführt, die der hier gewählten entspricht.*

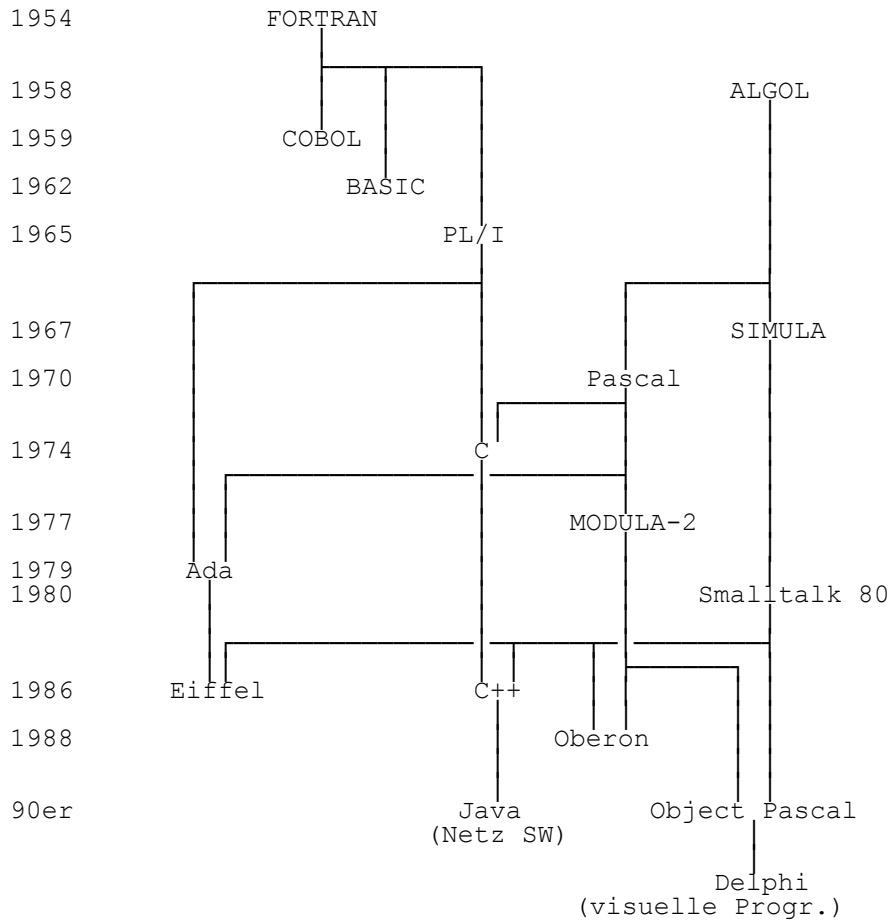
Wirth, N.: Algorithmen und Datenstrukturen. B. G. Teubner, Stuttgart 1983

1 While-Programme

1.1 Einstieg in C und erste Programme

Die Sprache C wird anhand von Beispielen vorgestellt. Die Semantik wird mittels Computer exploriert.

Stammbaum der wichtigsten höheren Programmiersprachen:



Aufgezeigt sind die zwei wichtigsten Entwicklungslinien der Programmiersprachen: die „amerikanische Linie“ mit den Sprachen FORTRAN und COBOL (IBM), und die von ALGOL ausgehende „europäische“. Nicht erfasst ist die „KI-Linie“ mit den Sprachen PROLOG, LISP und Miranda.

> Erhellend, was die Entwicklung der Programmiersprachen betrifft, ist der folgende Auszug aus dem von mir im ersten Semester benutzten Lehrbuch der Programmierung (Dieter Müller: Programmierung elektronischer Rechenanlagen. BI Mannheim 1966): „Trotz der großen Unterschiede in der Struktur der verschiedenen Rechenanlagen soll der Inhalt dieses Taschenbuches die praktische Herstellung arbeitsfähiger Programm ermöglichen. Die zunehmende Verbreitung problemorientierter Programmsprachen läßt einen solchen Versuch nicht völlig aussichtslos erscheinen. Wie die Tabelle 1 des Anhangs zeigt, können fast alle zur Zeit in Deutschland allgemein zugänglichen Rechenanlagen mit Hilfe einer der Programmsprachen FORTRAN oder ALGOL programmiert werden... Die zur Zeit an den verschiedenen Rechenanlagen benutzten FORTRAN-Systeme weichen noch so stark voneinander ab, dass eine gemeinsame Beschreibung sich entweder auf sehr wenige Sprachelemente beschränken oder eine große Zahl maschinenabhängiger Details enthalten muß.“ Die Tabelle 1 enthielt noch eine Aufstellung sämtlicher Rechner an Hochschulen. Nicht nur die Rechnerwelt war überschaubar, man wusste auch, welche Übersetzer wo vorhanden waren. In Darmstadt waren wir stolz auf den Compiler ALCOR Illinois. Programme wurden auf Lochkarten gestanzt: Je Zeile eine Karte. Der Kar-

tenstapel wurde im Vorraum des Rechenzentrums auf einem Gestell abgelegt. Von dort kam es in den Stapelbetrieb der Rechenanlage. Der Betrieb wurde von den DV-Spezialisten des Rechenzentrums kontrolliert und blieb uns weitgehend verborgen. Am nächsten Tag konnte man die Karten auf einem anderen Gestell wieder abholen, dazu das Ergebnis - oft nur eine ganze Serie von Fehlermeldungen in den - für Typenraddrucker typischen - tanzenden Buchstabenreihen auf Endlospapier. Es folgte die Korrektur des Programms in der Hoffnung, wenigstens am Tag darauf das Rechenergebnis in der Hand zu halten. (Sie wurde oft enttäuscht.) Das Verfahren bewirkte, dass man bei der Fehlersuche wesentlich sorgfältiger vorging als heutzutage.

Vom Quelltext zum ausführbaren Programm

Den *Quelltext* (source code) eines Programms kann mit jedem beliebigen Editor oder Textverarbeitungsprogramm erstellt werden. Wichtig ist nur, dass schließlich eine reine ASCII-Datei (Textdatei ohne weitere Formatierungsangaben) entsteht. Als Dateiendung wird der Buchstabe `c` gewählt, um sie als C-Programm kenntlich zu machen (zum Beispiel: `TelefonListe.c`).

Einfache Programm werden einfach dadurch in ein lauffähiges Programm übersetzt, dass man hinter den Aufruf des C-Compilers (`BCC32` im Falle des Borland 32-Bit-C-Compilers) den Dateinamen anhängt. Es entsteht dann eine ausführbare Datei (executable file) mit der Dateiendung `.exe`. Der Aufruf

```
Bcc32 TelefonListe.c
```

erzeugt demnach die ausführbare Datei `TelefonListe.exe`. Gibt man nun den Namen `TelefonListe` ein, wird das Programm gestartet.

Ein erstes Programm in C

Ein Programm ist eine Sammlung von *Deklarationen* und *Funktionsdefinitionen*, von denen eine den Namen `main` besitzt. Der *Rumpf* der Funktionsdefinition besteht aus einer *zusammengesetzten Anweisung* (compound statement); das ist eine Folge von Deklarationen und *Anweisungen* (statements), die in geschweifte Klammern eingeschlossen sind. Eine zusammengesetzte Anweisung wird auch *Block* genannt. *Kommentare* beginnen mit /* und enden mit */.

Im einfachsten Fall besteht ein Programm nur aus der Funktionsdefinition `main` – wie im folgenden Beispiel einer einfachen Umrechnung von inch in cm:

```
/*Ein-Aus.C*/
#include <stdio.h>
main() {
  int inch;
  printf("? inch = ");
  scanf("%d", &inch);
  printf("!  %d in = %.3f cm\n", inch, inch*2.54);
  return 0;
}
```

Erläuterungen zum Programm: Der Anfangskommentar enthält nur den Namen, unter dem die Datei abgespeichert ist: „Ein-Aus.C“. Dieser Kommentar hat keinerlei Einfluss auf die Funktion des Programms. Die Endung „.C“ wollen wir grundsätzlich für C-Quelltexte reservieren.

Dass `main` eine Funktion ist, erkennt man an den nachfolgenden runden Klammern. Ohne weitere Bestimmung ist der Rückgabewert eine ganze Zahl. Wir hätten das auch explizit machen können durch Angabe des Typs `int` vor dem Namen `main`. Die Wertübergabe interes-

siert uns nicht weiter. Der Rückgabewert wird deswegen - zur Vermeidung von Warnungen durch den Übersetzer - einfach auf null gesetzt. Das geschieht mit der letzten Anweisung, nämlich „`return 0;`“.

C gehört zu den *typisierten Programmiersprachen*. Die Typisierung ist eine wichtige Eigenschaft moderner Programmiersprachen, die dem Programmierer hilft, Fehler zu vermeiden. Bereits der Übersetzer kann nämlich ein Programm auf die falsche Verwendung von Variablen überprüfen.

Also: Die Variablen eines C-Programms müssen vor der ersten Verwendung deklariert sein. Die zusammengesetzte Anweisung des Rumpfes beginnt demzufolge mit der Deklaration der später verwendeten Variablen `inch`. Es handelt sich um eine Variable vom Typ *Integer* (*ganze Zahl*), wie das Schlüsselwert `int` erkennen lässt.

Es folgen drei Anweisungen, die - wie die Deklaration - jeweils von einem Semikolon abgeschlossen werden. Die erste Anweisung ist eine *Ausgabeanweisung*. Sie hat den Namen `printf`. Dass es sich um eine Funktion handelt, ist an den nachfolgenden runden Klammern erkennbar, die gegebenenfalls eine Parameterliste umfassen. Der erste Parameter der Print-Anweisung ist die *Zeichenkette* (*Textstring*) `"? inch = "`. Diese wird unverändert auf dem Bildschirm ausgegeben.

Es folgt die *Leseanweisung* `scanf` - ebenfalls eine Funktion. Wie die Ausgabeanweisung hat sie als ersten Parameter einen Textstring. Dieser sagt hier nur, welches *Format* das durch die Anweisung von der Tastatur zu lesende Datum `inch` hat. Formatangaben beginnen mit einem %-Zeichen. Das nachfolgende `d` besagt, dass über die Tastatur eine Ganzzahl in Dezimaldarstellung einzugeben ist. Der zweite Parameter der Leseanweisung ist `&inch`. Das ist die Adresse der Variablen `inch`. Sie sagt, wo das eingegebene Datum abzuspeichern ist. Das &-Zeichen (Ampersand) heißt Adressoperator.

Die letzte Anweisung ist wieder eine Ausgabeanweisung. Der erste Parameter ist wiederum ein Formatstring, der neben dem direkt auszugebenden Text noch Formatangaben für die auszugebenden Daten enthält. Die auszugebenden Daten sind die nachfolgenden Parameter der Anweisung. Die erste Formatangabe `%d` besagt, dass an dieser Stelle der Wert von inch als ganze Dezimalzahl auszugeben ist. Die zweite, nämlich `%.3f`, sorgt dafür, dass der letzte Paramter der Anweisung, nämlich `inch*2.54` als Fixpunktzahl mit drei Stellen nach dem Dezimalpunkt auf dem Bildschirm erscheinet. Anstelle der Formatangaben erscheinen die entsprechend formatierten Daten auf dem Bildschirm.

In den obigen Beispielen kommen nur sogenannte *automatische Variable* vor, das sind Variable, die in einer Funktion deklariert und nicht ausdrücklich als statisch deklariert sind. Sie werden zur Laufzeit lokal zu dem Block angelegt, in dem sie definiert sind und nicht implizit *initialisiert*. Demgegenüber existieren die *statische Variablen* während der gesamten *Laufzeit* eines Programms. Diese werden implizit zu null initialisiert. Statische Variable sind entweder außerhalb von Funktionen oder ausdrücklich als statisch deklariert.

Ein allgemeiner Hinweis zur Durchführung von Übungen am Computer

Anfangs wird man zur Übung nur vorgegebene Programmtexte abtippen und versuchen zum Laufen zu bringen. Tut man's einfach so, ohne dabei groß nachzudenken, wird man nicht viel dabei lernen. Für einen möglichst großen Lerneffekt in der Anfangsphase sollte man sich an die folgenden Regeln halten:

1. Schreiben Sie das Programm und übersetzen Sie es. Korrigieren Sie dabei eventuelle Tippfehler. Lassen Sie das Programm noch nicht laufen.

2. Machen Sie eine *Prognose:* Notieren Sie sich dazu die von ihnen beabsichtigten Eingaben und notieren Sie das von ihnen erwartete *Resultat* des Programmlaufs. Schreiben Sie auf, was auf dem Bildschirm erscheinen wird.
3. Starten Sie das Programm, geben Sie die Daten wie notiert ein.
4. *Vergleichen* Sie ganz genau das vom Computer gelieferte Resultat mit Ihrer Prognose.
5. *Analysieren* Sie die Abweichungen und ziehen Sie Ihre Lehren daraus.
6. Variieren Sie die Eingabe und verfahren Sie wie oben: Prognose, Vergleich und Analyse.
7. Wiederholen Sie das so lange, bis Sie sicher sind, die Funktion des Programms zu kennen.
8. Variieren Sie den Programmtext und führen Sie erneut die Schritte Prognose, Vergleich und Analyse aus.

Etwas Standard-Ein-/Ausgabe für Zahlen

Einige wichtige Bezeichner für *Argumenttypen*:

d: Ganzzahl in Dezimaldarstellung
u: nichtnegative Ganzzahl in Dezimaldarstellung
c: Zeichen vom Typ char
e, g: Gleitpunktzahl
f: Fixpunktzahl

Das Ausgabeformat von Gleitpunktzahlen kann folgendermaßen genauer festgelegt werden:

```
Formatangabe = % [MinimaleFeldweite][. Genauigkeit]Argumenttyp
Argumenttyp = e | f | g
```

Zu Beginn des Einführungskurses in eine Progammiersprache noch ein Rat, eine Art übergeordnete *Programmierregel:* Vorbilder nutzen. Programmtexte der Meister der Programmierung (Dahl, Dijkstra, Hoare, Kernighan, Knuth, Mills, Ritchie, Wirth u.a.) lesen und analysieren. Alternative Lösungen entwickeln und versuchen, die Vorbilder zu übertreffen.

Übungen

1.1.1 Schreiben Sie die Ausgabeanweisung für eine Zeile, in der die ganze Zahl i (hier gleich 25), und die Gleitpunktzahlen x (hier gleich 0.5) und x (hier gleich 3.1415…) folgendermaßen als Tabellenzeile erscheinen sollen:

```
_25|_0.500|_3.142
```

Der Unterstrich steht für ein Leerzeichen. Hinter dem letzten Zeichen (hier eine 2) erfolgt ein Zeilenvorschub. Bei den Gleitpunktzahlen sollen bis zu zwei Stellen vor dem Dezimalpunkt darstellbar sein.

1.2 Lexikalische Elemente und einfache Datentypen

Der Zeichensatz von C

Der Zeichensatz von C besteht aus den alphanumerischen Zeichen (Buchstaben, Ziffern) dem Unterstrich (underscore _) sowie den Trennungs und den *Sonderzeichen* (siehe Tabelle).

Die alphanumerischen Zeichen sind:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

a b c d e f g h i j k l m n o p q r s t u v w x y z

0 1 2 3 4 5 6 7 8 9

Tabelle der Sonderzeichen

Zeichen	englischer Name	Zeichen	englischer Name
,	comma	!	exclamation mark
.	period	|	vertical bar
;	semicolon	/	forward slash
:	colon	\	backslash
?	question mark	~	tilde
'	single quotation mark	+	plus sign
"	double quotation mark	#	number sign
(	left paranthesis	%	percent sign
)	right paranthesis	&	ampersand
[	left bracket	^	caret
]	right bracket	*	asterisk
{	left brace	-	minus sign
}	right brace	=	equal sign
<	left angle bracket	>	right angle bracket

Bezeichner (auch: Namen, englisch: Identifier), Konstanten und Schlüsselwörter bestehen aus den alphanumerischen Zeichen sowie dem Unterstrich. Groß- und Kleinbuchstaben werden unterschieden (case sensitive). Erster Buchstabe muss ein Buchstabe oder der Unterstrich sein.

Trennungszeichen (white-space characters) sind SP (Zwischenraum, englisch: space), die *Steuerzeichen* HT, VT, LF, CR, FF, und die Kombination für den Beginn einer neuen Zeile (new line) CR LF. Für die Darstellung der Steuerzeichen bedient man sich der *Escape-Sequenzen* der folgenden Tabelle. Kommentare werden vom Compiler wie Trennungszeichen behandelt. Trennzeichen stehen zwischen den lexikalischen Elementen - das sind die bedeutungstragenden Einheiten oder Token - und niemals innerhalb derselben.

Tabelle der Escape-Sequenzen

Escape-Sequenz	Kurzzeichen	deutscher Name	englischer Name
\a	BEL	Klingel	bell (alert)
\b	BS	Rückwärtsschritt	backspace
\f	FF	Formularvorschub	form feed
\n	LF CR	Neue Zeile (Zeilenvorschub und Wagenrücklauf)	new line
\r	CR	Wagenrücklauf, Eingabe	carriage return
\t	HT	Horizontaltabulator	horizontal tab
\v	VT	Vertikaltabulator	vertical tab
\\	\	Rückwärtsschrägstrich	backslash
\?		Fragezeichen	question mark
\'		Apostroph (einfaches Anführungszeichen)	single quotation mark
\"		doppeltes Anführungszeichen	double quotation mark
\ooo			ASCII character (octal notation)
\xhh			ASCII character (hexadezimal notation)
\0	NUL	Der Wert 0, das Nullzeichen	Null character

Darüberhinaus gehören auch Sonderzeichen wie @, $, und § zum darstellbaren Zeichensatz.

Konstante

Konstante (constant) *ganze Zahlen* werden folgendermaßen dargestellt:

Ziffernfolge	Dezimale Angabe	(Ziffern: 0-9)
0Ziffernfolge	Oktale Angabe	(Ziffern: 0-7)
0xZiffernfolge	Hexadezimale Angabe	(Ziffern: 0-9, a-f, A-F)
0XZiffernfolge	Hexadezimale Angabe	

Anmerkung: Bei der dezimalen Angabe darf die erste Ziffer nicht Null sein. Bei der oktalen Angabe muß die erste Ziffer Null sein. Die einzelnen Ziffern der Konstanten dürfen nicht durch Trennungszeichen (white-space) getrennt werden. Die Konstanten haben immer positive Werte. Negative Werte werden dadurch gebildet, dass vor die Konstante ein Minuszeichen geschrieben wird. Durch den unären Minusoperator wird die Konstante zu einem konstanten Ausdruck. Der Typ der Konstanten hängt von ihrem Wert ab. Es wird jeweils der erste Typ der folgenden Tabelle gewählt, der eine Darstellung des Wertes erlaubt.

Tabelle der Konstantentypen

Form	möglicher Typ
dezimal	int, long int, unsigned long int
oktal, hexadezimal	int, unsigned int, long int, unsigned long int
Suffix: l, L	long int, unsigned long int
Suffix: u, U	unsigned int, unsigned long int
Suffix: ul, uL, Ul, UL, lu, Lu, lU, LU	unsigned long int

Fest- und *Fließpunktkonstante* sind folgendermaßen definiert:

```
floating-point_constant = fractional_constant [exponent_part]
                          [type_suffix] |
                          digit {digit} exponent_part [type_suffix]
fractional_constant =     {digit} "." digit {digit} |
                          digit {digit} "."
exponent_part = e_or_E [plus_or_minus] digit {digit}
e_or_E = "e" | "E"
plus_or_minus = "+" | "-"
type_suffix = "f" | "F" | "l" | "L"
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

Beispiele: Handelt es sich bei den folgenden Zahlen um Fließpunktkonstanten (floating-point constant) nach obiger Syntaxdefinition?

```
1.2E-3
.E+2
1.E+18F
2.2E
1.234 E -3
```

Zeichen-Konstanten sind in Apostrophe eingeschlossene Zeichen. Beispielsweise steht 'a' für den Buchstaben a. ' ' steht für ein Leerzeichen und '\'' ist die Zeichen-Konstante für das Zeichen ' (Apostroph). Escape-Sequenzen zur Zeichendarstellung sind erlaubt. Der Wert der Zeichen-Konstanten entspricht dem numerischen Wert des Zeichens in der ASCII-Tabelle. Zeichen-Konstanten haben den Typ int (Vorzeichenexpansion !).

Zeichenfolgen-Konstanten sind in Anführungszeichen eingeschlossene Zeichenfolgen. Beispiele: "Dies ist eine \"Zeichenfolgen-Konstante\"\n". Die Verkettung von Zeichenfolgenkonstanten geschieht durch Hintereinanderschreibung.

Zur rechnerinternen Speicherung von Zeichenfolgen (Strings): Alle Zeichen einer Zeichenfolge werden in aufeinanderfolgenden Speicherzellen abgelegt. Das Ende der Zeichenfolge wird durch das Zeichen \0 (binäre Null) gekennzeichnet, das automatisch angehängt wird und damit kein Zeichen der Zeichenfolge sein darf (nullterminierte Strings). Zeichenfolgen-Konstanten haben den Typ char[], wobei das Feld um ein Zeichen größer ist als die Zeichenfolgen-Konstante selbst (für das Zeichen \0).

Es ist nicht sichergestellt, dass zwei identische Zeichenfolgen-Konstanten denselben Speicherplatz belegen (i.a. wird für jede Konstante ein eigener Speicherbereich benutzt). Es ist aber auch nicht sichergestellt, dass zwei identische Zeichenfolgen-Konstanten verschiedene Speicherplätze belegen. Programme sollten keine Modifikationen an einer Zeichenfolgen-Konstanten vornehmen.

Bezeichner

Bezeichner (identifier) bestehen aus Buchstaben (A-Z, a-z, _) und Ziffern (0-9). Das erste Zeichen muss ein Buchstabe sein. Der Unterstrich zählt zu den Buchstaben. Es wird zwischen Groß- und Kleinbuchstaben unterschieden. Bezeichner werden für Variable, Konstanten, Typen, Funktionen oder Marken (label) verwendet. Bezeichner müssen sich von Schlüsselwörtern unterscheiden. Bezeichner, die mit einem Unterstrich beginnen, werden intern im Compiler und in der Standard-Bibliothek verwendet. Der Anwender sollte sie meiden.

Die folgenden *Schlüsselwörter* haben für den Compiler eine besondere Bedeutung:

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

Kommentare

Kommentare (comment) beginnen mit „/*“ und enden mit „*/“. Jeder Kommentar wird vom Compiler wie ein einzelnes Trennungszeichen (white-space character) behandelt. Kommentare dürfen an jeder Stelle des Programms stehen, an der Trennungszeichen erlaubt sind. Kommentare dürfen nicht geschachtelt werden.

Token

Ein Token ist die Basiseinheit, die ein Compiler in einem Programm erkennt. Operatoren, Schlüsselwörter oder Bezeichner sind Token. Token werden begrenzt durch Trennungszeichen oder andere Token. Der Compiler versucht möglichst viele Zeichen zu einem Token zusammenzufassen, bevor er ein neues Token beginnt.

Einfache Datentypen, Array-Typ und Pointer-Arithmetik

Die wichtigsten fundamentalen Datentypen sind die ganzahligen Typen `char, short, int` und `long`. Die Datentypen für reelle Zahlen heißen `float` und `double`. Wird den ganzzahligen Typen das Schlüsselwörtern `unsigned` beigefügt, handelt es sich um nichtnegative Zahlen. Genauer werden diese Typspezifikatoren (type-specifiers) erst in späteren Abschnitten behandelt.

Der Wert einer *Zeiger-* oder *Pointer-Variablen* ist die Adresse einer Speicherzelle eines bestimmten Typs oder aber der *Nullpointer* - geschrieben als 0 oder `NULL`. Ist `v` Variable eines bestimmten Typs, dann ist `&v` die zugehörige Adresse (Pointer- oder Zeiger-Wert). Ist umgekehrt `w` eine Pointervariable, dann bezeichnet `*w` die Variable, auf die der Pointer zeigt (Dereferenzierung)[1].

---

[1] Im Abschnitt über den Stack (Informatik I) hatten wir es schon einmal mit dieser Unterscheidung von Zeigern und Variablen zu tun: Die Speicherzelle SP war dort ein Pointer, nämlich der Stackpointer. Als Zeichen der Dreferenzierung haben wir dort die runden Klammern verwendet: (SP) entsprach der Variablen TOP. Nach C-Konvention schreibt man anstelle der Klammern ein Sternchen vor die Pointervariable: *SP.

Zwischen den Variablen der Mathematik und denen der Informatik besteht ein wesentlicher Unterschied: In der Mathematik steht eine Variable für einen beliebigen aber fest gewählten Wert - mathematische Variablen sind sozusagen Stellvertreter für Werte. In der Informatik ist eine Variable ein Behälter für einen Wert. Der Wert selbst ist veränderbar, also wirklich variabel.

Beispiele für *Variablendeklarationen*:

```
(1)int i;
(2)float x, y;
(3)char c1 = 'a', *p = &c1, c2 = *p, *q;
(4)char a[80], b[80], *p, *q;
(5)char v[2][5] = {{'a', 'b', 'c', 'd', 'e'},
                   {'0', '1', '2', '3', '4'}};
```

Die Deklarationen (1) und (2) machen die *Variable* i zu einer ganzen Zahl und x und y zu Gleitpunktzahlen einfacher Genauigkeit. Die Deklaration (3) definiert die Variable c1 als Zeichen und initialisiert sie auf einen Wert, nämlich den Buchstaben a. Ferner wird p als Zeiger (Pointer) auf ein *Zeichen* (*character*) definiert. „char *p“ kann man auch so lesen: „Die Dereferenzierung von p ist eine Integer-Variable“. Der Pointer p wird auf die Adresse von c1 initialisiert. Also bezeichnen anschließend c1 und *p dieselbe Variable. Folglich sind auch die Werte von *p und c1 gleich; c2 ist eine weitere Variable, die anfangs denselben Wert wie c1 und *p zugewiesen bekommt. Und q ist ein nicht initialisierter Zeiger auf eine Zeichenvariable. Durch die Deklaration (4) werden a und b zu Arrays aus jeweils 80 Zeichen. Die Zeichen des Arrays a sind beispielsweise a[0], a[1], a[2], ..., a[79]. Und p und q sind wieder Pointer auf Zeichen.

Das *Array* v gemäß Deklaration (5) hat die Elemente v[0][0], v[0][1], v[0][2], v[0][3], v[0][4], v[1][0], v[1][1], v[1][2], v[1][3], v[1][4]. Sie sind in genau dieser Folge im Speicher hintereinander und nach steigenden Adressen untergebracht. Durch die Initialisierung gilt beispielsweise v[1][3] == '3'. Falls die Initialisierungsliste vorzeitig abbricht, wird der Rest zu null initialisiert.

Der Array-Bezeichner (hier v) ist synonym zum Zeiger (pointer) auf das erste Objekt (Element) des Arrays. Anstelle von &v[0] kann man also auch einfach v schreiben.

Typisch für C ist die sogenannte Pointer-Arithmetik (Kernighan/Ritchie, S. 100-103): Verweist ein Zeiger auf ein Objekt eines Arrays, dann darf man eine ganze Zahl addieren. Die Summe p+n eines solchen Zeigers p und einer ganzen Zahl n ist ein Zeigerwert, der auf das n-te Objekt hinter dem ursprünglichen zeigt. Das heißt: Bei der Pointer-Arithmetik wird stets der Objekttyp und dessen Speicherbedarf eingerechnet. Bei der Adressberechnung ist also auf die ursprüngliche Adresse das n-fache der für ein Element des Arrays benötigten Speicherplätze zu addieren. Mit *(v+n) wird also dieselbe Variable bezeichnet wie mit v[n].

Es gibt nicht eigens einen booleschen Datentyp: Zahlenwerte und Pointer ungleich null werden mit der booleschen Konstanten wahr bzw. 1 identifiziert; und die boolesche Konstante falsch bzw. 0 ist gegeben durch die Zahl 0 bzw. den Nullpointer.

Interpretation von Zeichen

Im Zusammenhang mit der Ein- und Ausgabe von Zeichenfolgen über Tastatur und Bildschirm (Standard I/O) treten folgende prinzipielle Schwierigkeiten auf:

Die *Formatierung* von Zeichenfolgen (die Markierung von Anfang und Ende beispielsweise) ist mit Zeichen aus demselben Zeichensatz darzustellen wie die Zeichenfolge selbst. Es gibt die folgenden Methoden:

1. Die Anführungszeichen beispielsweise werden als *Sonderzeichen* behandelt, die ansonsten im Text nicht auftreten dürfen. Beispiel: "Gibt's das?". Will man auch das Anführungszeichen darstellen, wählt man ein alternatives Sonderzeichen, z. B. den Apostroph: 'Er sagte: "Heute nicht."'. So wird's in Modula 2 gemacht.
2. Methode der *Zeichen-Einfügung* (Einführung in die Informatik, Teil 2): In Pascal ist das Begrenzungszeichen für Zeichenketten der Apostroph. Kommt der Apostroph in der Zeichenkette selbst vor, wird ein weiterer Apostroph eingefügt. Beispiel: 'Gibt"s das?'.
3. *Escape-Sequenzen:* In C werden Zeichen-Konstanten durch Anführungszeichen begrenzt. Kommt das Anführungszeichen im Text vor, ist ein Backslash (\) voranzustellen. Beispiel: "Er sagte: \"Heute nicht.\"". Auch zur Darstellung des Backslashs wird diese Methode verwendet: "\\\\" steht für zwei Backslashs in Folge.

Nichtdarstellbare Zeichen: In Verbindung mit der numerischen Tastatur lassen sich Zeichen der Codetabelle - siehe „Einführung in die Informatik 1“, Abschnitt 1.2 „Binäre Codierung“ - direkt über ihre Ordnungszahl eingeben: Mit <Alt> 102 wird beispielsweise das Zeichen „f“ eingegeben. Für die Steuerzeichen wird die Steuerungstaste zusammen mit den Buchstaben des Alphabets benutzt: <Strg> C wirkt genauso wie <Alt> 3, da C der dritte Buchstabe des Alphabets ist. Für die Eingabe von Steuerzeichen stehen damit u. a. die folgenden Tastaturcodes zur Verfügung:

End of Text = ETX = <Strg> C = <Alt> 3
Backspace = BS = <Strg> H = <Alt> 8
Linefeed = LF = <Strg> J = <Alt> 10
Carriage Return = CR = <Strg> M = <Alt> 13
Substitute = SUB = <Strg> Z = <Alt> 26
Escape = ESC = <Alt> 27

Das Substitutionszeichen SUB wird oft als Dateiendezeichen (EOF) verwendet (Beispielsweise beim Standardstrom cin). Für den Rückwärtsschritt (BS), die Codeumschaltung (ESC) und den Wagenrücklauf (CR) gibt es eigene Tasten. CR löst - je nach Eingabefunktion - mehrere Wirkungen aus. Beispielsweise die folgenden.

1. Markierung des Endes der Folge
2. Auslösen der Übergabe einer gepufferten Folge
3. Zeilenvorschub und Anhängen des LF-Zeichens an die Folge
4. Der Cursor wird auf die ersten Position der Zeile gesetzt (Wagenrücklauf)

Beim Bildschirm-Echo werden manche der nichtdruckenden Zeichen durch besondere Symbole dargestellt - z. B. `^Z` für SUB; bei anderen wird die durch das Zeichen codierte Aktion ausgeführt.

Eingabefunktionen wie get haben Filterfunktion und fügen eigene Formatierungszeichen ein. Z.B.: Löschen der nichtdruckenden Zeichen und Einfügen eines Zeichens für das Textende. Eingabefunktionen lesen die Zeichen der Tastatur über einen Puffer ein - oder auch ungepuffert. Die Lesefunktionen sorgen gegebenenfalls auch für ein Echo der eingegebenen Zeichen auf dem Bildschirm - oder sie unterdrücken es.

Übungen

1.2.1 Was wird auf dem Bildschirm erscheinen, wenn Sie das folgende C-Programm laufen lassen? Erläutern Sie, was passiert. Testen Sie Ihre Prognosen am Computer.

```
/*Dekl.C*/
#include <stdio.h>
main(){
```

```
   char c1 = 'a', *p = &c1, c2 = *p, *q;
   q = &c2;
   printf("Demo: Zeiger und Adressen\n*q = %c\n", *q);
  }
```

Die include-Anweisung sorgt dafür, dass die Deklarationen von Variablen und Funktionen, die in einer anderen Datei definiert sind, eingefügt werden. Im Fall der Standardein- und Standardausgabe stdio geht es um die Eingabefunktion scanf und die Ausgabefunktion printf.

1.2.2 Was wird auf dem Bildschirm erscheinen, wenn Sie das folgende C-Programm laufen lassen?

```
/*Array.C*/
#include <stdio.h>
main(){
 char z[2][3] = {{'a', 'b', 'c'}, {'0', '1', '2'}},
      *p=*z, *q=*(z+1), *r=*(z+1)+2;
 printf("char z[2][3] = {{'a', 'b', 'c'}, {'0', '1', '2'}},\n");
 printf("    *p=*z, *q=*(z+1), *r=*(z+1)+2;\n");
 printf("Ausgabe als Char:\n");
 printf("!: z[0][0]: %c\n", z[0][0]);
 printf("!: **z: %c\n", **z);
 printf("!: z[1][1]: %c\n", z[1][1]);
 printf("!: *(*(z+1)+1): %c\n", *(*(z+1)+1));
 printf("!: (*(z+1))[1]: %c\n", (*(z+1))[1]);
 printf("!: *p: %c\n", *p);
 printf("!: *q: %c\n", *q);
 printf("!: *r: %c\n", *r);
 printf("Ausgabe als Int:\n");
 printf("!: z: %d\n", z);
 printf("!: *z: %d\n", *z);
 printf("!: **z: %d\n", **z);
 return 0;
}
```

1.3 Syntax und Semantik von Ausdrücken

Die Sprache C hat ein außerordentlich mächtige Grammatik für die Formulierung von Ausdrücken. Die Beherrschung der Syntax kennzeichnet den guten C-Programmierer. Die Mächtigkeit des Systems hat einen großen Nachteil: der Lernaufwand ist ziemlich groß. Die Sprache bietet nicht nur exzellente Möglichkeiten, das Richtige zu sagen, sondern auch das Falsche. Die Sprache C hat einiges zu bieten, was Denkfallen und Gelegenheiten für Programmierfehler angeht.

Vor der Syntaxdarstellung der Ausdrücke werden in dieser Lektion Beispiele gebracht und erläutert. Wer sich die Sprache C erschließen will, sollte unbedingt alle diese Ausdrücke nach der Syntaxdefinition zergliedern (parsen) und den zugehörigen Syntaxbaum zeichnen.

Gute Fertigkeiten bei der Programmkonstruktion wird nur erwerben, wer sich an guten Vorbildern orientiert. Lesen Sie die Programme der Meister der Programmierung und untersuchen Sie die Programme hinsichtlich der C-Syntax. Orientieren Sie sich bei der Konstruktion von Programmen grundsätzlich an der Syntaxdarstellung. Geben Sie sich nicht mit der naheliegendsten Lösung eines Problems zufrieden. Suchen Sie anhand der Syntaxdarstellung nach Alternativen. Sie werden sich wundern, welche Möglichkeiten in der Sprache stecken - nicht zu vergessen die Fehlermöglichkeiten!

Zuweisungsausdrücke und arithmetische Ausdrücke

Der Ausdruck „a=2“ ist ein Zuweisungsausdruck. Er bewirkt, dass nach Auswertung des Ausdrucks die Variable a den Wert 2 hat. Auf der rechten Seite des Zuweisungszeichens kann ein beliebiger (mit a zuweisungsverträglicher) Ausdruck stehen. Er kann selbst wieder ein Zuweisungsausdruck sein. Beispielsweise bewirkt der Ausdruck „a=b=c=2“, dass nacheinander den Variablen c, b und a der Wert 2 zugewiesen wird. Der Ausdruck selbst hat den Wert und den Typ der Variablen ganz links.

Ein Zuweisungsausdruck muss nicht unbedingt ein Zuweisungszeichen „=„ enthalten.

Zur Bildung arithmetischer Ausdrücke stehen das Pluszeichen „+“, das Minuszeichen „-“, das Multiplikationszeichen „*“ und das Divisionszeichen „/“ zur Verfügung. Mittels runder Klammern „(“ und „)“ können Teilausdrücke zusammengefasst werden. Die üblichen Regeln für den Vorrang der Operatoren und für das Setzen von Klammern sind gültig. Beispielsweise hat der Ausdruck „(2+3*4)*5“ den Wert 70.

Durch Komma (Sequenzoperator) voneinander abgetrennte Zuweisungsausdrücke werden von links nach rechts ausgewertet. Der Wert des gesamten Ausdrucks ist gegeben durch den letzten Zuweisungsausdruck der Folge. Die Auswertung des Ausdrucks „a=2, b=3, b=b*4, (a+b)*5“ bewirkt, dass zuerst die Variable a den Wert 2 zugewiesen bekommt, dann erhält die Variable b den Wert 3, danach wird der Wert von b ersetzt durch den mit 4 multiplizierten alten Wert 3 der Variablen b, und schließlich wird der Zuweisungsausdruck „(a+b)*5“ berechnet. Er hat den Wert 70. Und dieser Wert ist auch der Wert des gesamten Ausdrucks.

Die Berechnung und Ausgabe des Wertes eines Polynoms $a+bx+cx^2$ könnte man mit folgender Anweisung erledigen:

```
printf("! a+bx+cx^2: %e\n", a+b*x+c*x*x);
```

Mit dem *Horner-Schema* geht es auch so

```
printf("! a+bx+cx^2: %e\n", a+(b+c*x)*x);
```

Verrückt aber lehrreich ist folgende Anweisung, die - abgesehen von der Veränderung des Variablenwerts von c - dasselbe Ergebnis hat:

```
printf("! a+bx+cx^2: %e\n", (c=c*x, c=b+c, c=c*x, a+c));
```

Lehrreich ist das Beispiel deshalb, weil hier ein Problem von C sehr deutlich wird: In C haben einige Symbole mehrere Bedeutungen, beispielsweise das Komma. Einerseits trennt es die Parameter eines Funktionsaufrufs voneinander ab, andererseits ist es Sequenzoperator in Ausdrücken. Deshalb *muss* in der obigen Anweisung der Ausdruck in runden Klammern stehen.

In C gibt es neben dem Zuweisungsoperator „=„ auch noch solche, die das Wiederholen von Variablennamen bei Anweisungen der Form „c=c*x“ vermeiden. Man schreibt für den genannten Ausdruck beispielsweise „c*=x“. Die Polynomausauswertung lässt sich unter Ausnutzung der verschiedenen Zuweisungsoperatoren folgendermaßen schreiben:

```
printf("! a+bx+cx^2: %e\n", (c*=x, c+=b, c*=x, a+c));
```

Eine Schreibweise mit Operatoren, die Verknüpfung und Zuweisung kombinieren, hat Vorteile, wenn die Variable selbst über einen Ausdruck ermittelt werden muss. Typischerweise kommt das bei Arrays vor, zum Beispiel im Ausdruck „v[i+1]“. Der Ausdruck „i=1, v[2]=2, v[i+1]+=3“ bewirkt, dass die Variable v[2] schließlich den Wert 5 hat.

Einen Ausdruck, dessen Ergebnis eine Variable ist, bezeichnet man als Lvalue. Jeder Variablenbezeichner ist ein solches Lvalue. Aber auch der Ausdruck v[i+1] ist ein Lvalue. Lvalues dürfen auf der linken Seite eines Zuweisungsausdrucks stehen. Daher kommt auch der Name: L steht für left.

Da das Erhöhen einer ganzen Zahl um +1 und das Verringern um -1 häufig vorkommen, gibt es in C die unären Operatoren „++“ und „--“. Die Ausdrücke „++i“ und „i++“ bewirken jeweils eine Erhöhung des Wertes von i um +1. Sind diese Ausdrücke wiederum nur Teil eines größeren Ausdrucks, dann spielt es eine Rolle, ob ein solcher Operator vorgestellt ist oder nachgestellt. Ist der Operator vorgestellt, wird die Veränderung des Variablenwertes vorgenommen, bevor die Auswertung weiter geht. Andernfalls wird der alte Wert genommen und die Erhöhung erst anschließend durchgeführt. Analoges gilt für „--“. Beispielsweise hat der Ausdruck „i=3, i++“ den Wert 3. Als Nebenwirkung wird erreicht, dass i anschließend den Wert 4 hat. Der Ausdruck „i=3, --i“ hat den Wert 2, genauso wie die Variable i im Anschluss an die Auswertung.

Wahrheitswerte und logische Ausdrücke

Wie bereits erwähnt, werden Zahlenwerte und Pointer ungleich null mit der booleschen Konstanten wahr bzw. 1 identifiziert. Die boolesche Konstante falsch bzw. 0 ist gegeben durch die Zahl 0 bzw. den Nullpointer.

Die Ausdrücke der Form

```
„a<b“, „a<=b“, „a>b“, „a>=b“, „a==b“, „a!=b“
```

heißen Vergleichsausdrücke. Anstelle der Variablen a und b können auch kompliziertere Ausdrücke stehen. Die Bedeutungen erschließen sich aus den Symbolen. Beispielsweise steht „<=„ für „kleiner oder gleich“. Das Doppelgleichheitszeichen steht für „gleich“. Dieser Handstand ist nötig, weil das Gleichheitszeichen bereits für die Zuweisung verbraucht ist; „!=„ steht für „ungleich“. Wenn der Vergleich wahr ist, hat der Vergleichsausdruck den Zahlenwert 1, ansonsten ist er gleich 0.

Logische Ausdrücke - also Ausdrücke, deren Ergebnisse als logisch 0 oder logisch 1 interpretiert werden können - lassen sich mit den Booleschen Operatoren verküpfen: „||“ ist die OR-Verknüpfung, „&&“ die AND-Verknüpfung und „!“ die Negation. Grundsätzlich wird bei der

Auswertung solcher Ausdrücke das Kurzschlussverfahren verwendet. Der Ausdruck „1<2||1/x“ hat den Wert 1, auch wenn x den Wert 0 hat und damit der zweite Teilausdruck undefiniert ist.

Im Unterschied zu den logischen Operatoren wirken die Operatoren „|“, „&“ und „^“ - letzterer steht für das XOR - bitweise. Will man alle bis auf die letzten vier Bits der ganzen Zahl a auf null setzen, kann man das beispielsweise so tun: „a=a&16-1“, oder auch so „a&=15“.

Im Zusammenhang mit den bitweisen Operationen werden oft Verschiebeoperationen benötigt. Für die Verschiebung von Bitmustern stehen die Operatoren „>>„ und „<<„ zur Verfügung: Der Ausdruck „i=1, z<<++i“ liefert ein Bitmuster, bei dem die Bits gegenüber z um zwei Speicherstellen nach links, also hin zu den höherwertigen Stellen, verschoben sind. Das entspricht einer Multiplikation der Zahl mit der Zahl 4, vorausgesetzt, auf den beiden höchstwertigen Stellen stehen Nullen.

Der bedingte Ausdruck

Mit Teilausdrücken a, b und c kann man den bedingten Ausdruck „a?b:c“ hinschreiben. Er hat folgende Bedeutung: Vorausgesetzt wird, dass der Wert des Ausdrucks a sich als Wahrheitswert interpretieren lässt. Ist dieser Wert ungleich null (TRUE), dann hat der bedingte Ausdruck den Wert des Teilausdrucks b, ansonsten den des Teilausdrucks c. Damit kann man beispielsweise den Absolutwert einer Zahl z folgendermaßen ausdrücken: „z<0?-z:z“.

Die Syntaxdefinition der Ausdrücke (expressions)

Die Komplexität des Ausdruckssystems von C entsprechend, ist die Syntaxdefinition recht umfangreich. Terminalen Symbole werden entweder in Apostrophe eingeschlossen, oder aber fett geschrieben.

Die Bedeutung der Operatoren und eine weitere Darstellung der Vorrangregelung für Ausdrücke (expression) sind der Tabelle am Schluss des Unterabschnitts zu entnehmen.

```
expression = assignment-expression { "," assignment-expression }

assignment-expression = conditional-expression |
   unary-expression assignment-operator assignment-expression

assignment-operator = "=" | "*=" | "/=" | "%=" | "+=" | "-=" | "<<=" |
   ">>=" | "&=" | "^=" | "|="

conditional-expression = logical-OR-expression |
   logical-OR-expression ? expression : conditional-expression

logical-OR-expression =
   logical-AND-expression { || logical-AND-expression }

logical-AND-expression =
   inclusive-OR-expression { && inclusive-OR-expression }

inclusive-OR-expression =
   exclusive-OR-expression { | exclusive-OR-expression }

exclusive-OR-expression = AND-expression { ^ AND-expression }

AND-expression =
   equalitiy-expression { & equalitiy-expression }
```

```
equalitiy-expression =
  relational-expression { equalOp relational-expression }

equalOp = "==" | "!="

relational-expression =
  shift-expression { relOp shift-expression }

relOp = "<" | ">" | "<=" | ">="

shift-expression =
  additive-expression {shiftOp additive-expression }

shiftOp = "<<" | ">>"

additive-expression =
  multiplicative-expression { AddOp multiplicative-expression }

AddOp = "+" | "-"

multiplicative-expression =
  cast-expression { MultOp cast-expression }

MultOp = "*" | "/" | "%"

cast-expression =
  unary-expression | ( type-name ) cast-expression

unary-expression = postfix-expression | "++" unary-expression |
  "--" unary-expression | unary-operator cast-expression |
  sizeof unary-expression | sizeof ( type-name ) |

unary-operator = "&" | "*" | "+" | "-" | "~" | "!"

postfix-expression = primary-expression |
  postfix-expression [ expression ] |
  postfix-expression ( [ argument-expression-list ] ) |
  postfix-expression.identifier |
  postfix-expression -> identifier |
  postfix-expression "++" |
  postfix-expression "--"

primary-expression =
  identifier | constant | string | ( expression )

argument-expression-list =
  assignment-expression { , assignment-expression }

constant = integer-constant | character-constant |
  floating-constant | enumeration-constant
```

Hinweise: Die Konstanten sind im Abschnitt 1.2 besprochen worden.

Unäre Operatoren und Zuweisungsoperatoren sind rechtsassoziativ. Alle andern sind linksassoziativ: a=b=c ist bedeutungsgleich mit a=(b=c) und a+b+c ist bedeutungsgleich mit (a+b)+c.

Die Syntaxdarstellung legt fest, in welcher Reihenfolge die Verknüpfungen geschehen. Sie sagt aber wenig über die Auswertungsreihenfolge von Unterausdrücken aus. Diese bleibt im allgemeinen undefiniert; die Sprachdefinition lässt da dem Compilerbauer gewisse Freiheiten.

Dieser Tatbestand zusammen mit der Tatsache, dass der C-Programmierer gern von Nebenwirkungen der C-Ausdrücke Gebrauch macht, ist eine der Fallen, die diese Sprache bietet.

Nach dem Gesagten ist folgende Anweisung syntaktisch zulässig aber semantisch fragwürdig bzw. unsinnig: „v[i] = i++“. Hier bleibt offen, ob zuerst die Variable v[i] bestimmt wird, oder ob zunächst der Teilausdruck i++ behandelt wird. Der Wert der rechten Seite ist in beiden Fällen gleich, aber die Feldauswahl ist verschieden. Im zweiten Fall würde die Feldauswahl v[i] gegenüber dem ersten Fall um ein Feld weiter rücken.

Lvalue wird ein Ausdruck genannt, der ein Objekt (Speicherbereich, Variable) bezeichnet. In Zuweisungsausdrücken der Form $expr_1$ = $expr_2$ muss $expr_1$ ein Lvalue sein. Semantik: Der Ausdruck $expr_2$ wird ausgewertet und das Ergebnis wird der durch $expr_1$ bezeichneten Variablen zugewiesen. Wert und Typ des gesamten Zuweisungsausdrucks sind schließlich gleich dem Wert und dem Typ von $expr_1$.

Runde Klammern werden in folgenden Bedeutungen verwendet: Funktionsaufrufoperator, Typ-Umwandlung, Vorrangregelung, Funktionsdefinition.

Die Operatoren „,“, „&&“ und „||“ garantieren die Auswertung von links nach rechts. Die logischen Operatoren werden darüberhinaus nach dem Kurzschlussverfahren ausgewertet und wirken deshalb wie CAND und COR (siehe Teil I).

Die Operanden der bitweisen Operationen müssen ganzzahlig sein.

Beispiel 1: Setzen Sie Klammern in den folgenden Ausdrücken so, dass sich die Bedeutung nicht ändert:

```
(1)a+b*c
(2)*p++
```

Beispiel 2: Wie werden die folgenden Ausdrücke interpretiert?

```
(1)i<=0 || max<i
(2)0<=i<100
(3)*q++ = *p++
```

Wie muss der zweite Ausdruck umgeformt werden, so dass das erreicht wird, was der Programmierer offenbar wollte?

Beispiel 3: Für welche Werte der nichtnegativen ganzen Zahl a ist der folgende boolesche Ausdruck wahr? Welchen Wert hat die Variable nach Auswertung des Ausdrucks?

```
a++<=1
```

Übungen

1.3.1 Zeichnen Sie die Syntaxbäume der folgenden Ausdrücke aus und beschreiben Sie die deren Funktion.

```
1) p2->sign=1
2) i = Beta*(c - c_old)
3) *q++ = *p++
4) 0<j & x[j]<x[j-1]
```

1.3.2 Erläutern Sie die Funktion des folgenden Programms. (Die Funktion gets(...) liest einen String, der über Tastatur eingegeben wird. Die Funktion sscanf(...) liest dann diesen String, so wie scanf(...) direkt von der Tastatur liest.)

```
/*Ausdruck.C*/
void main(){
  char z; int i; float x, y;
  char s[80]; /*Puffer für eine Bildschirmzeile*/
  printf("? Eingabe (char, int, float, float): "); gets(s);
  printf("! Echo1: %s\n", s);
  sscanf(s, "%c,%d,%e,%e", &z, &i, &x, &y);
  printf("! Echo2: %c  %d  %e  %e\n", z, i, x, y);
```

```
  printf("! Summe: %e\n", z+i+x+y);
}
```

1.3.3 Das folgende Programm berechnet den Betrag einer komplexen Zahl *x*+j*y*. Es benutzt im Vorgriff die Funktionen fabs(...) und sqrt(...) der mathematischen Standardbibliothek. Diese bilden den Absolutwert einer Gleitpunktzahl bzw. deren Quadratwurzel. Das ausgedruckte Ergebnis ist gleich $\sqrt{x^2 + y^2}$. Zeigen Sie, dass der im Programm gewählte Ausdruck äquivalent zu dieser Formel ist. Finden Sie heraus und erläutern Sie, warum die Formel nicht direkt realisiert worden ist. Denken Sie an mögliche Overflow- und Underflow-Fehler.

```
/*Betrag.C*/
#include <stdio.h>
#include <math.h>
main(){
  float x, y;

  printf("? Eingabe x y: ");
  scanf("%e%e", &x, &y);
  printf("! Echo: %e  %e\n", x, y);
  printf("! Betrag = %e\n",
    fabs(x)<fabs(y)? fabs(y)*sqrt(1+(x/y)*(x/y))
    : x==0? 0: fabs(x)*sqrt(1+(y/x)*(y/x))
  );
  return 0;
}
```

Tabelle Operatoren

Die Operatoren sind nach fallendem Vorrang geordnet. Zwischen den Trennstrichen stehen Operatoren gleichen Vorrangs.

Operator	**Name und Bedeutung**	**Syntax**	**Beispiel**
.	Memberauswahl	*Objekt.Member*	(*p).x
->	Memberauswahl	*Pointer -> Member*	p->x
[ ]	Feldindizierung	*Pointer* [*Ausdr*]	a[i]
( )	Funktionsaufrufsoperator	*Ausdr* (*Ausdr-Liste*)	f(x)
++	Postinkrementierung	*Lvalue*++	*q++=*p++
--	Postdekrementierung	*Lvalue*--	
sizeof	Typgröße	sizeof (*Typ*)	in Vielfachen von char
++	Preinkrementierung	++*Lvalue*	++i
--	Predekrementierung	--*Lvalue*	
~	Einerkomplement	~*Ausdr*	
!	Negation	!*Ausdr*	!0 ist gleich 1
-	unäres Minus	-*Ausdr*	
+	unäres Plus	+*Ausdr*	
&	Adressoperator	&*Lvalue*	
*	Dereferenzierung	**Ausdr*	char *p=&c1
( )	Typ-Umwandlung (Type Cast)	(*Typ*) *Ausdr*	(point*)pred
*	Multiplikation	*Ausdr* * *Ausdr*	
/	Division	*Ausdr* / *Ausdr*	
%	Modulo	*Ausdr* % *Ausdr*	
+	Addition	*Ausdr* + *Ausdr*	a+b+c
-	Subtraktion	*Ausdr* - *Ausdr*	
<<	Verschiebe nach links um	*Ausdr*<<*Ausdr*	
>>	Verschiebe nach rechts um	*Ausdr*>>*Ausdr*	
<	kleiner	*Ausdr*<*Ausdr*	
<=	kleiner oder gleich	*Ausdr*<=*Ausdr*	
>	größer	*Ausdr*>*Ausdr*	
>=	größer oder gleich	*Ausdr*>=*Ausdr*	
==	gleich	*Ausdr*==*Ausdr*	
!=	ungleich	*Ausdr*!=*Ausdr*	

Tabelle Operatoren (Fortsetzung)

Operator	Name und Bedeutung	Syntax	Beispiel
&	bitweise AND	*Ausdr*&*Ausdr*	
^	bitweise EXOR	*Ausdr*^*Ausdr*	
|	bitweise OR	*Ausdr*|*Ausdr*	
&&	CAND	*Ausdr*&&*Ausdr*	
||	COR	*Ausdr*||*Ausdr*	
? :	bedingter Ausdruck	*Ausdr*?*Ausdr*:*Ausdr*	
=	einfache Zuweisung	*Lvalue*=*Ausdr*	a=b=b+c
*=	multiplizieren und zuweisen	*Lvalue**=*Ausdr*	
/=	dividieren und zuweisen	*Lvalue*/=*Ausdr*	
%=	modulo und zuweisen	*Lvalue*%=*Ausdr*	
+=	addieren und zuweisen	*Lvalue*+=*Ausdr*	a=b+=c
-=	subtrahieren und zuweisen	*Lvalue*-=*Ausdr*	
<<=	nach links verschieben und zuweisen	*Lvalue*<<=*Ausdr*	
>>=	nach rechts verschieben und zuweisen	*Lvalue*>>=*Ausdr*	
&=	AND und zuweisen	*Lvalue*&=*Ausdr*	
|=	OR und zuweisen	*Lvalue*|=*Ausdr*	
^=	EXOR und zuweisen	*Lvalue*^=*Ausdr*	
,	Sequenzoperator (Komma)	*Ausdr*, *Ausdr*	

1.4 Syntax und Semantik der While-Programme

Die Syntax von C-Anweisungen

Die Syntax der *Anweisungen* (statements) der Programmiersprache C ist hier nahezu vollständig wiedergegeben. Auf das Instrumentarium der freien Sprünge mittels Goto wurde verzichtet. Die Syntax bietet unübersehbare Ausdrucksmöglichkeiten. In den folgenden Unterabschnitten werden Grundstrukturen vorgestellt, die nur eine Teilmenge der Grammatik benötigen. Es handelt sich um eine leicht beherrschbare Teilmenge, die zum Grundbestand der *strukturierten Programmierung* gehört (Dahl/Dijkstra/Hoare, 1972): neben der Zuweisung sind das die grundlegenden *Strukturtypen* Sequenz, Auswahl, und Schleife. Das Strukturtheorem (structure theorem) von Linger/Mills/ Witt (1979) zeigt, dass die Vielfalt der realisierbaren Funktionen nicht beschränkt wird, wenn man allein diese grundlegenden Strukturtypen verwendet.

```
statement = expression-statement | compound-statement |
            selection-statement | iteration-statement |
            labeled-statement | jump-statement

labeled-statement =
  case constant-expression : statement | default : statement

constant-expression = conditional-expression

expression-statement = [ expression ] ;

compound-statement =
  { [ declaration-list ] [ statement-list ] }

declaration-list = declaration { declaration }

statement-list = statement { statement }

selection-statement =
  if ( expression ) statement [ else statement ] |
  switch ( expression ) statement

iteration-statement =
  while ( expression ) statement |
  do statement while ( expression ) ; |
  for ([expression]; [expression]; [expression]) statement

jump-statement = continue ; | break ; | return [ expression ] ;
```

Semantik von Anweisungen: Bezeichnungen

Das Paradigma der *imperativen Programmierung* hat Niklaus Wirth im Titel eines seiner Bücher formuliert: Algorithms + Data Structures = Programs. Anweisungen, die den Algorithmus ausmachen, sind von den eigentlichen Datenstrukturen und Variablen fein säuberlich getrennt. Die Bedeutung einer Anweisung ist durch die Wirkung gegeben, die sie auf die Werte der Variablen hat.

Wir bezeichnen mit V die Menge der deklarierten Variablen und mit W die Vereinigung sämtlicher Wertebereiche dieser Variablen. Ein *Zustand* (oder eine Wertebelegung) σ ist definiert als Funktion

$\sigma: V \rightarrow W$

Die Menge aller Zustände nennen wir Σ. Und mit σ[*x*/*a*] bezeichnen wir denjenigen Zustand, der sich vom Zustand σ höchstens im Wert der Variablen *x* unterscheidet. Es gilt σ[*x*/*a*](*x*) = *a* und σ[*x*/*a*](*y*) = σ(*y*) für alle *y*≠*x*.

Eine Funktion *f*: $A \rightarrow B$ nennen wir partiell, wenn zugelassen wird, dass sie für Werte $x \in A$ auch undefiniert ist. Unter der Bedeutung (Semantik) einer Anweisung *S* verstehen wir eine *partielle Funktion*

$S: \Sigma \rightarrow \Sigma$

Diese Funktion beschreibt die Wirkung, die die Anweisung *S* auf die Variablenwerte hat.

Die Zuweisung

Die Anweisungen wirken auf die Variablen, indem sie deren Werte ändern. Im einfachsten Fall geschieht das durch eine *Zuweisungsanweisung* (*assignment statement*) oder kürzer: Zuweisung. Eine Zuweisung ist ein expression-statement, der folgenden Grundstruktur

x= *e*;

Sie besteht also aus einem Zuweisungsausdruck (assignment-expression), der durch ein abschließendes Semikolon zu einer Anweisung wird. Anstelle von *e* kann eine Variabel stehen, oder - allgemeiner - ein Zuweisungsausdruck[1]. Die Bedeutung der Zuweisung ist folgendermaßen definiert: Es ist

„x = *e*;“(σ) = σ[x/*e*(σ)]

für alle $\sigma \in \Sigma$, für die *e*(σ) definiert ist[2]. Das gilt unter der Bedingung, dass der Ausdruck *e* keine weiteren Nebenwirkungen entfaltet.

In Worten: Durch eine Zuweisung wird nur der Wert der Variablen verändert, die auf der linken Seite der Anweisung steht. Deren neuer Wert ist schließlich gleich dem Wert des Ausdrucks auf der rechten Seite - ausgewertet im alten Zustand.

Beispiel: Seien Alpha und Beta die deklarierten Variablen vom Typ int. Wir betrachten die Wirkung der Zuweisungsanweisung „Alpha = (Alpha + Beta)/2;“ auf den Zustand σ, der gegeben ist durch (Alpha, Beta) = (3, 14). Es ergibt sich: σ' = „Alpha = (Alpha + Beta)/2;“(σ). Das Divisionszeichen / wird hier, aufgrund der ganzzahligen Argumente, im Sinne der ganzzahligen Division (Zeichen: div) aufgefasst[3]. Der Wert des Ausdrucks (Alpha + Beta)/2 im Zustand σ ist gleich (3+14) div 2, also gleich 8. Folglich ist der schließlich sich ergebende Zustand σ' gegeben durch σ'(Alpha) = 8 und σ'(Beta) = 14. Die Funktion der Zuweisung ist undefiniert, wenn sich der Ausdruck auf der rechten Seite nicht auswerten lässt, zum Beispiel bei einer Division durch null.

---

[1] Zur gewählten Darstellung: Wenn innerhalb eines Programms etwas kursiv geschrieben ist, dann handelt es sich um den Stellvertreter für einen noch näher zu bestimmenden Text, ähnlich einem nichtterminalen Symbol der Syntaxbeschreibung.

[2] In der obigen mathematischen Relation wird die Funktion durch das in Anführungszeichen gestellte Programm selbst repräsentiert. Man beachte: Das Gleichheitszeichen hat im mathematischen Ausdruck tatsächlich die Bedeutung "gleich". Im Programm ist die Bedeutung eine andere, nämlich "wird zugewiesen der Wert des Ausdrucks".

[3] Hier haben wir es übrigens mit einem der Fälle zu tun, in denen ein Operatorzeichen mit mehreren Bedeutungen "überladen" ist: Einmal wirkt es wie das normale Divisionszeichen und ein andermal als Zeichen für die ganzzahlige Division: 1/2 ist also nicht etwa gleich 0.5 sondern gleich null. Das kann schnell zu schwer diagnostizierbaren Programmierfehlern führen.

Übung: Zu den vielen Möglichkeiten, Ausdrücke mit Nebenwirkungen zu erzeugen, gehört die geschachtelte Zuweisung: „x = y = x-y;“. Hier wird nicht nur x, sondern auch y geändert. Auch so etwas ist möglich: „x = x + (y = y+1);“, oder „x = x+ ++y“. Ermitteln Sie jeweils die Bedeutung der Anweisungen. Beschreiben Sie, welche Schwierigkeiten solche Konstruktionen im Rahmen größerer Programme machen können.

Programmierregel: Vermeiden Sie Nebenwirkungen. Drücken Sie Ihre Absichten explizit aus.

Die Sequenz

Die Bedeutung einer Anweisungsfolge

$S_1\ S_2\ ...\ S_n$

ergibt sich aus der Hintereinanderausführung der jeweiligen Funktionen der einzelnen Anweisungen:

„$S_1\ S_2\ ...\ S_n$“$(\sigma) = S_n(...(S_2(S_1(\sigma))))$

Also: Zuerst wird die Anweisung S_1 auf den Zustand σ angewendet und es ergibt sich der Zustand $\sigma_1 = S_1(\sigma)$. Auf diesen Zustand wird dann die Anweisung S_2 angewendet, so dass sich der Zustand $\sigma_2 = S_2(\sigma_1)$ ergibt, usw. Schließlich erhält man mit $\sigma_n = S_n(\sigma_{n-1})$ das Resultat der Anweisungsfolge. Die Bedeutung einer Verbundanweisung (compound statement) ist gleich der Bedeutung der in ihr enthaltenen Anweisungsfolge.

Eine Anweisungsfolge aus i identischen Anweisungen „$S\ S\ ...\ S$“ bezeichnen wir mit S^i. S^0 ist die leere Anweisung mit der Bedeutung $S^0(\sigma) = \sigma$.

Beispiel: Die Anweisungsfolge „y= x*2; x=x+y;“ liefert für den Anfangszustand (x, y) = (2, 3) wegen (2 , 4) = „y= x*2;“(2, 3) und (6, 4) = „x=x+y;“(2 , 4) das Ergebnis (6, 4).

Die Auswahl

Die Bedeutung der Auswahl

if (B) S_1 else S_2

wird definiert durch

„if (B) S_1 else S_2“$(\sigma) = S_1(\sigma)$, falls $B(\sigma)$ ungleich null (also wahr) ist,

und

„if (B) S_1 else S_2“$(\sigma) = S_2(\sigma)$, falls $B(\sigma)$ gleich null (also falsch) ist.

Beispiel: Im Zustand (x, y) = (1, -2) liefert die Anweisung `if (x<y) x= y; else y= x;` den Zustand (x, y) = (1, 1).

Die Schleife

Mit diesen Bezeichnungen lässt sich die Bedeutung der Schleife

while (B) S

folgendermaßen definieren: Sei n die kleinste nichtnegative Zahl, für die $\neg B(S^n(\sigma))$ gilt, dann ist

„while (B) S“$(\sigma) = S^n(\sigma)$

In folgenden Fällen ist die Schleife nicht definiert:

- $B(S^i(\sigma))$ ist für alle i wahr. Dann bricht die Berechnung nie ab (Endlosschleife).
- Im Verlaufe der Berechnungen entsteht ein Zustand, für den entweder B oder S nicht definiert ist.

Beispiel 1: Gegeben ist das Programm

```
/*Min.C*/
#include <stdio.h>
#include <stdlib.h>

int x[5]= {2, 3, 0, 1, 2};
void main(){
 int i= 0, min=x[0];
 printf("Min von 2, 3, 0, 1, 2\n");
 while(i<4) if(x[++i]<min) min= x[i];
 printf("! min= %d", min);
}
```

Welchen Wert nimmt min schließlich an?

Bevor wir den Programmablauf durchgehen, bringen wir die vorletzte Anweisung des Programms noch in eine äquivalente und leichter überschaubaren Form:

```
while(i<4) {++i; if(x[i]<min) min= x[i];}
```

Wir beschreiben die Zustände durch Vektoren in mathematischer Schreibweise (n-Tupel):

```
(x[0], x[1], x[2], x[3], x[4], min, i)
```

Man prüft leicht nach, dass unmittelbar vor Eintritt in die Schleife der Zustand gleich

```
(2, 3, 0, 1, 2, 2, 0)
```

ist. Die Schleifenbedingung ist erfüllt. Durchlaufen der Schleife liefert nacheinander die Zustände

```
(2, 3, 0, 1, 2, 2, 1)
(2, 3, 0, 1, 2, 0, 2)
(2, 3, 0, 1, 2, 0, 3)
(2, 3, 0, 1, 2, 0, 4)
```

Nun ist erstmals die Schleifenbedingung nicht erfüllt und die Schleife bricht ab. Die Variable min hat schließlich den Wert der kleinsten Komponente des Arrays x, nämlich null.

Beispiel 2: Swap. Gesucht ist die Funktion (Bedeutung, Semantik) der Anweisungsfolge

```
t = x;
x = y;
y = t;
```

Dabei spielt t nur die Rolle einer Hilfsvariablen.

Wir benennen die Anfangswerte der Variablen mit kursiven Großbuchstaben, denn es handelt sich um Variablen im Sinne der Mathematik: (x, y, t) = (X, Y, T). Nun schreiben wir die Zustandsfolge auf, die sich beim Programmablauf ergibt:

```
(X, Y, T)
(X, Y, X)
(Y, Y, X)
(Y, X, X)
```

Ergebnis: Die Anweisungsfolge vertauscht die Werte der Variablen x und y.

Beispiel 3: Eine While-Schleife im Pascal-Stil:

```
/*Schleife.C*/
#include <stdio.h>
main(){
  char a[80], b[80], *p, *q;
  int i = 0;
  printf("Demo: Schleife\n? Text  = "); gets(a);
  while (a[i]!=0) {b[i]=a[i]; i=i+1;}
  b[i]=a[i];
  printf("! Pascal-Stil\n Kopie = %s\n", b);
}
```

So sieht die Schleife in einem Stil aus, der halb Pascal und halb C ist:

```
p = a, q = b;
while (*p != 0) {*q = *p; p = p+1; q = q+1;}
*q = 0;
```

Und C-typisch ist dagegen das folgende Programmfragment, das dieselbe Aufgabe löst:

```
p = a, q = b;
while (*q++ = *p++);
```

Übung: Vergleichen Sie die Programme hinsichtlich Verständlichkeit.

Weitere Wiederholungs- und Auswahlanweisungen

Das Continue-Statement erscheint nur in Iteration-Statements und beendet den aktuellen Schleifendurchlauf (statement). Das Break-Statement erscheint nur in Iteration- oder Switch-Statements (Kernighan/Ritchie, S. 58-65). Es beendet dieses. Beispielsweise kann man den Abbruch einer Schleife in den Schleifenkörper hinein verlagern: „while(1){... if (!*B*) break; ...}“.

Das Return-Statement beendet eine Funktion und liefert den Rückgabewert - falls verlangt.

Das Switch-Statement hat die Grundgestalt

```
switch (expression) {
   case const-expr1: statements1
   case const-expr2: statements2
   ...
   default: statements
}
```

Die Bearbeitung startet mit dem Fall, dessen *const-expr* mit dem Wert von *expression* übereinstimmt. Auch die darauf folgenden Fälle werden ausgeführt, solange bis ein Break- oder Return-Statement erscheint. Typischerweise werden also sämtliche Statements von ein Break-Statement abgeschlossen.

Die Anweisung „for(*a*, *b*, *c*) *S*“ ist äquivalent zu „*a*; while(*b*) {*S c*;}“. Die For-Anweisung erlaubt es also, alle Steueranweisungen einer Schleife zusammenzufassen, die Anweisung „*c*;“ dient der Änderung der Steuervariablen der Schleife. Die Äquivalenz der beiden Schleifenanweisungen gilt mit der Einschränkung, dass in der Schleifenanweisung *S* keine Continue-Anweisung vorkommt. Enthält S eine Continue-Anweisung und wird diese tatsächlich ausgeführt, so wird im Falle der For-Anweisung im Anschluss an *S* auch noch *c* ausgeführt, im Falle der While-Schleife jedoch nicht.

Eine Schleife der Gestalt „for(;;) {...}“ kann nur von einer Break- oder einer Return-Anweisung im Inneren des Schleifenrumpfs (Verbundanweisung) beendet werden, ansonsten handelt es sich um eine Endlosschleife.

Wenn die Schleifenanweisung zwingend ein erstes Mal durchlaufen werden soll, kann man anstelle einer While-Schleife „*S* while(*B*) *S*“ auch eine Do-while-Schleife wählen: „do *S* while(*B*);“.

Regeln zur Textgestaltung

Die *Textgestaltung* ist ein wirkungsvolles Mittel zur Verdeutlichung der Steuerungsstruktur eines Programmes. Die ersten der folgenden Programmierregeln sind allgemein verbindlich. Die übrigen Regeln können auch anders aussehen. Diese hier betreffen speziell die C-Programmierung.

1. Den Programmtext nicht künstlerisch, sondern nach Regeln gestalten.
2. Jede Deklaration oder Anweisung steht vorzugsweise auf einer eigenen Zeile.
3. Kompakt schreiben, denn der Bildschirm soll Information zeigen.
4. Die öffnende geschweifte Klammer „{„ startet den Einzug (und steht somit am Zeilenende). Der von den Klammern eingeschlossen Block beginnt – mit Einzug – auf der folgenden Zeile. Mit der zugehörigen schließende Klammer „}“ wird dieser Einzug wieder aufgehoben. Sie steht allein auf der Zeile.
5. Einzugstiefe: maximal 2 Stellen (das reicht und passt zur 2. Regel).
6. Einzelne Ausdrucksanweisungen in Auswahlanweisungen oder Schleifenanweisungen, die auf einer eigenen Zeile stehen, werden ebenfalls eingezogen, z. B.:

```
if (x<y)
  x= y;
else
  y= x;
```

7. Kurze Verbund-, Schleifen- oder Auswahlanweisungen dürfen auch in einer Zeile stehen.
8. Unnötigen Einzug vermeiden. Mit Leerzeilen sparsam umgehen.

Gute Textgestaltung mittels *Einzug* (Indentation) gehört zu den Lernzielen eines Einführungskurses in die Programmierung. Sie macht graphische Darstellungmittel wie die *Struktogramme* nach *Nassi-Shneiderman* oder dergleichen überflüssig.

Übungen

1.4.1 Schreiben Sie das folgende Programm nach den Regeln der Textgestaltung neu:

```
void main() {int i, j; srand(time(0));
if ((textfile=fopen("Lager1.dat", "w"))==NULL) {
printf("\nFehler: Datei speichern nicht m”glich"); exit(0);}
fprintf(textfile, "Lager1: %d Versuche mit je %d Stichproben", m, n);
fprintf(textfile, "\nVersuch\tm-E\tm\tm+E");
for (i=0; i<m; i++) {double L=0, M=0, S=0, E;
for (j=0; j<n; j++) {L=a()-b(); if(L<0) L=0; M+=L; S+=L*L;}
M/=n; E=t*sqrt((S/n-M*M)/(n-1)); fprintf(textfile, "\n%d\t%e\t%e\t%e",
i, M-E, M, M+E);} fclose(textfile);}
```

1.5 Die Invariante

Was dem Naturwissenschaftler die Naturgesetze, das sind dem Informatiker die Invarianten: Die Invariante erfasst den wesentlichen Kern eines Algorithmus, sie ist sein „Gesetz".

Erik – wir kennen ihn aus der ersten Lektion – bildet er nacheinander die Zahlenfolgen

(202)
(40, 2)
(8, 0, 2)
(1, 3, 0, 2)

Jede dieser Dezimalzahlen wird bei Erik durch einen Haufen Steine repräsentiert. Aber darauf kommt es jetzt nicht an. Dahinter stecken folgende Gleichungen:

$202 = 202 \cdot 5^0$
$202 = 40 \cdot 5^1 + 2 \cdot 5^0$
$202 = 8 \cdot 5^2 + 0 \cdot 5^1 + 2 \cdot 5^0$
$202 = 1 \cdot 5^3 + 3 \cdot 5^2 + 0 \cdot 5^1 + 2 \cdot 5^0$
$202 = 0 \cdot 5^4 + 1 \cdot 5^3 + 3 \cdot 5^2 + 0 \cdot 5^1 + 2 \cdot 5^0 = 1 \cdot 5^3 + 3 \cdot 5^2 + 0 \cdot 5^1 + 2 \cdot 5^0$

Das sind die Schritte, mit denen Erik die Zahl $z = 202$ in die Darstellung im Stellenwertsystem zur Basis $b = 5$ wandelt. Wir nummerieren die Zeilen von oben nach unten durch: $k = 0, 1, 2, \ldots$ Die Gleichung mit der Nummer k lässt sich mit einer geeigneten Zahlenfolge $(z_k, c_{k-1}, ..., c_1, c_0)$ so darstellen:

$$z = z_k \cdot b^k + c_{k-1} \cdot b^{k-1} + ... + c_1 \cdot b^1 + c_0 \cdot b^0 \qquad (*)$$

In unserem Fall sind die Zahlenwerte konkret gegeben durch

$z_0 = z = 202$, $z_1 = 40$, $z_2 = 8$, $z_3 = 1$, $z_4 = 0$

$c_0 = 2$, $c_1 = 0$, $c_2 = 3$, $c_3 = 1$

Das Wesen dieses Algorithmus kommt darin zum Ausdruck, dass sich für jeden seiner Schritte, also für $k = 0, 1, 2, \ldots$, die folgende *Aussage* formulieren lässt: Für die Zahlenfolge $(z_k, c_{k-1}, ..., c_1, c_0)$ des Schrittes k gilt die Gleichung (*) und jede der Zahlen $c_{k-1}, ..., c_1, c_0$ ist kleiner als b; z_k darf auch größer sein. (Alle Zahlen werden hier grundsätzlich als nichtnegativ vorausgesetzt.)

Eine solche Aussage über die im Algorithmus vorkommenden Größen nennen wir *Prädikat*[1].

Ein Prädikat, das vor und nach jedem Arbeitsschritt eines Algorithmus gilt, ist eine *Invariante* dieses Algorithmus.

Eriks Aufgabe ist, unter Aufrechterhaltung der Invarianten, die Variable k Schritt für Schritt um je eins zu erhöhen und die Werte z_k immer kleiner zu machen, bis die *Endebedingung* $z_k = 0$ und damit die gewünschte Zahlendarstellung zur Basis b erreicht ist: $z = c_{k-1} \cdot b^{k-1} + ... + c_1 \cdot b^1 + c_0 \cdot b^0$.

---

[1] In der Aussagenlogik betrachtet man nur die Verknüpfung von Aussagen ohne Berücksichtigung deren innerer Struktur. Das Prädikatenkalkül geht über die Aussagenlogik hinaus. Ihr Gegenstand sind Aussagen über Subjekte. Die Aussage „$a<b$" betrachten wir in der Aussagenlogik als solche und setzen dafür eine Aussagenvariable ein. Wollen wir aber etwas über die möglichen Werte (Erfüllungsmengen) der (Zahlen-)Variablen a und b wissen, müssen wir die Aussagen als „Aussagen über etwas" – in unserem Fall über die Subjektvariablen a und b – auffassen, als Prädikate also. In die Alltagssprache übersetzt: „Hans geht nach Hause" ist eine Aussage. „Hans" ist das Subjekt und „… geht nach Hause" das Prädikat.

Bei einem Algorithmus muss man zwischen den *Handlungsanweisungen* (auch: *Operatoren*, oder auch nur: *Anweisungen*) und den Invarianten unterscheiden. Letztere dienen der vollständigen Beschreibung der Gesetzmäßigkeiten des Algorithmus.

Wenn die Operatoren dazu führen, dass Schritt für Schritt die anfangs gültigen Invarianten beibehalten werden, und wenn die Operatoren dafür sorgen, dass schließlich die Endebedingung erfüllt ist, dann sind die Operatoren geeignet zur Lösung der gestellten Aufgabe. Vorausgesetzt ist dabei, dass die Invariante tatsächlich das Wesentliche der Aufgabe vollständig erfasst.

Algorithmen haben oft die folgende Grundstruktur:

A
`while(`*B*`)`*S*

Dabei steht *A* für eine Anweisungen oder eine Folge von Anweisungen; *B* steht für einen Ausdruck, dessen Zahlenergebnis als wahr (≠0) oder falsch (=0) interpretiert wird - also so, als wäre er ein boolescher Ausdruck; und *S* steht für eine Anweisung. Letztere ist meist eine Verbundanweisung.

Unter einer *Schleifeninvarianten* wollen wir eine Invariante verstehen, die nach jedem Schleifendurchlauf gilt, vorausgesetzt sie gilt auch vor Eintritt in die Schleife. Die *Initialisierung A* hat als einzige Aufgabe, diese Invariante wahr zu machen, also dafür zu sorgen, dass die Invariante vor Eintritt in die Schleife gilt. Selbstverständlich darf man beim Beweis, dass die *Schleifenanweisung S* die Invariante erhält, die zusätzliche Vorbedingung *B* verwenden. Die Schleifeninvariante ist ein wichtiges Element des Algorithmenentwurfs.

Der Zusammenhang zwischen den Anweisungen und den Invarianten wird nun an einem einfachen Beispiel erläutert: Gegeben ist das folgende Programm.

```
/*Anzahl_p.C*/
#include <stdio.h>
#include <string.h>

main(){
 char x[80], a;
 int r, i, n;

 printf("? Textstring= "); gets(x);
 printf("? Buchstabe = "); a=getchar();
 n=strlen(x);

 r=0; i=0;
 while(i<n){if(x[i]==a) r++; i++;}

 printf("! Anzahl= %d", r);
 return 0;
}
```

Mit Hilfe der Schleifeninvarianten ist zu beweisen, dass das Programm ermittelt, wie oft ein bestimmter Buchstabe a in einem String x vorkommt. Zuerst formulieren wir geeignete „problemlösende“ Invarianten.

Wir setzen *f*(i) gleich der Anzahl der a-Werte unter den ersten i Werten des Arrays x; *f*(i) ist also gleich der a-Werte im Array-Abschnitt (x[0], x[1], ..., x[i-1]). Wir zeigen nun, dass r=*f*(i) eine Schleifeninvariante des obigen Programms ist. Ferner zeigen wir, dass auch i≤n eine Schleifeninvariante ist.

Zunächst betrachten wir den Initialisierungsteil „r=0; i=0;“ und prüfen, ob er tatsächlich die Invariante gültig macht.

Nach der Initialisierung ist i gleich 0 und r ebenfalls. Wir prüfen, ob die erste Schleifeninvariante gilt: Es ist $f(i) = f(0) = 0$, denn ein leerer Abschnitt des Arrays kann keine Wert enthalten, also auch keinen, der gleich a ist. Deshalb ist $r=f(i)$.

Falls $0 \le n$ ist, was wir voraussetzen dürfen, gilt nach dem Initialisierungsteil auch $i \le n$. Also sind nach der Initialisierung die beiden Invarianten wahr.

Nun zeigen wir, dass die Schleifenanweisung die Invarianten erhält, wobei wir zusätzlich annehmen dürfen, dass die Schleifenbedingung für die Anfangswerte gilt. Die Variablenwerte vor Durchlaufen der Schleifenanweisung bezeichenen wir wieder genauso wie die Variablen selbst. Die Werte nach Durchlaufen der Schleifenanweisung erhalten einen Strich. Genauer: Ist σ der Zustand des Rechners unmittelbar vor Eintritt in die Schleifenanweisung und σ' der Zustand unmittelbar nach Beendigung der Schleifenanweisung, und ist v eine Variable des Programms, dann bezeichnen wir mit v den Wert $v(\sigma)$, und mit v' den Wert $v(\sigma')$.

Offenbar dürfen wir voraussetzen[2], dass $r=f(i)$, $i \le n$ und $i<n$ gilt. Die Werte der Variablen nach dem Schleifendurchlauf sind $r'=r+1$ falls $x[i]=a$ und $r'=r$ sonst, sowie $i'=i+1$. Aus $i<n$ folgt $i' \le n$. Der Rest des Invarianzbeweises wird durch eine Fallunterscheidung erledigt:

1. Falls x[i] gleich a ist, gilt $r' = r+1 = f(i) + 1 = f(i+1) = f(i')$
2. Falls x[i] ungleich a ist, gilt $r' = r = f(i) = f(i+1) = f(i')$

Also in beiden Fällen gilt schließlich $r'=f(i')$. Damit ist die behauptete Invarianzeigenschaft bewiesen.

Da die Schleife abbricht, sobald i=n, liefert das Programm folgendes Resultat: i=n und $r=f(i)$, also schließlich: $r=f(n)$. Das heißt: Das Programm stellt fest, wievielmal der Wert a im Array x vorkommt und liefert diese Zahl auf der Variablen r ab.

Übrigens ist das Programm Anzahl_p.C im Pascal-nahen Stil geschrieben. Eher nach C sieht die folgende Version aus:

```
void main(){
 char x[80], *p, a;
 int r;

 printf("? Textstring= "); gets(x);
 printf("? Buchstabe = "); a=getchar();

 r=0; p=x;
 while(*p) if(*p++==a) r++;

 printf("! Anzahl= %d", r);
}
```

Programmierregel: Man lasse sich bei der Strukturierung eines Programmes vom **EVA**-Prinzip leiten: Nach der Deklaration und Initialisierung der Variablen eines Blockes kommt zunächst die **E**ingabe der Daten, dann die **V**erarbeitung und schließlich die **A**usgabe.

Übungen

1.5.1 Zeigen Sie allgemein, dass die Handlungsanweisung der Zahlenwandlung tatsächlich die Gültigkeit der Invarianten erhält. Nutzen Sie für den Beweis die Grundrelation der ganzzahligen Arithmetik: $x = (x \text{ div } y)\, y + x \bmod y$.

---

[2] In Programmbeweisen werden die Zeichen wie in der Mathematik üblich verwendet: "=" ist also das Gleichheitszeichen, und nicht etwa das Zuweisungszeichen der Programmiersprache.

1.5.2 Arbeiten Sie den Foliensatz „Invariante und Algorithmenentwurf“ (auf meiner Informatik-Web-Seite) durch. Notieren Sie auf einem Blatt Papier alles, was Ihnen Verständnisschwierigkeiten bereitet. Gehen Sie die Fragen in der Übung mit Ihrem Übungsleiter durch.

1.5.3 Gegeben ist ein Polynom $P(x) = c_n \cdot x^n + c_{n-1} \cdot x^{n-1} + ... + c_1 \cdot x + c_0$. Beweisen Sie, dass die Anweisungsfolge

```
j=n; r=c[j];
while(j>0)r=r*x + c[--j];
```

bei gegebenen Werten n, x und c[j] schließlich den Wert P(x) des Polynoms abliefert. Führen Sie zunächst eine Äquivalenztransformieren des Schleifenkörpers durch, so dass auf der rechten Seite von Zuweisungsausdrücken nur nebenwirkungsfreie Ausdrücke vorkommen.

Lösungshinweis: Die Invariante des Algorithmus ist $r = \Sigma_{0 \le i \le n-j} c_{j+i} x^i = c_j + c_{j+1} x^1 + c_{j+2} x^2 + ... + c_{n-1} x^{n-j-1} + c_n x^{n-j}$. Der Beweis Das geschieht in drei Schritten:

1. Nachweis, dass die Initialisierung die Invariante wahr macht
2. Nachweis, dass die Schleifenanweisung die Gültigkeit der Invarianten erhält
3. Nachweis, dass die negierte Schleifenbedingung zusammen mit der Invarianten das Resultat impliziert

Tip: Nehmen Sie ein einfaches allgemein geschriebenes Polynom, z. B. $c_0 + c_1 x + c_2 x^2 + c_3 x^3$, und gehen Sie dafür den Algorithmus Schritt für Schritt durch. Machen Sie sich so die Bedeutung der Invarianten klar. Finden Sie heraus, wie das hier angewandte Verfahren in der Mathematik genannt wird.

1.6 Suchen und Sortieren

Suchen in Arrays: Lineare Suche und Binäre Suche. Sortieren von Arrays: Insertion Sort. Der Wächter (Sentinel).

Suchen in Arrays

Suchproblem 1: Für ein Array, das mit der Konstanten n folgendermaßen deklariert ist

```
KomponentenTyp x[n];
```

soll bestimmt werden, auf welchem der Felder mit einem Index kleiner n erstmals der Wert a vorkommt. Also: der kleinste Index i ist gesucht, für den a gleich x[i] ist.

Eine einfache Möglichkeit zur Lösung des 1. Suchproblems ist die *lineare Suche:*

```
i=0;
while(i<n && a!=x[i]) i++;
```

Zwischenfragen: Wie sieht die Invariante des Algorithmus aus? Welche Bedingungen sind nach Ablauf des Algorithmus erfüllt? Was folgt, wenn schließlich i=n? Was folgt, wenn schließlich i<n?

Für i=n ist die Abfrage „a!=x[i]“ nicht definiert. Das schadet aber nichts, denn da die logischen Operatoren && und || nach dem Kurzschlussverfahren ausgewertet werden, findet diese Abfrage nie statt.

Störend an dem Algorithmus ist, dass bei jedem Schritt die Abfrage i<n anfällt. Sie wird für den Fall benötigt, dass das Element a im Array überhaupt nicht vorkommt und i über den letzmöglichen Index hinausschießt.

Durch eine kleine Modifikation machen wir den Algorithmus effizienter. Gleichzeitig vermeiden wir undefinierte Ausdrücke. Wir wählen das Array von vornherein etwas größer und hängen das gesuchte Element zusätzlich hinten an:

```
KomponentenTyp x[n+1];
```

Das Element x[n] mit dem Wert a heißt *Wächter* bzw. *Sentinel*. Jetzt kann man sich die Abfrage i<n sparen. Falls schließlich i=n, kommt das gesuchte Element auf den ersten n Plätzen nicht vor. Andernfalls ist i gleich dem Index, auf dem das Element erstmals auftritt.

Die *lineare Suche mit Sentinel:*

```
x[n]=a; i=0;
while(a!=x[i]) i++;
```

Es folgt als Beispiel ein Programm, das eine Folge von Strings entgegennimmt. Das Programm findet die Position eines bestimmten weiteren Wortes heraus. Jedes String wird zunächst auf eine Puffervariable a fester Länge gebracht. Für ein String wird dann jeweils nur so viel Speicherplatz reserviert, wie unbedingt nötig. Das geschieht mit dem Aufruf „`malloc(strlen(a) + 1)`“. Das Argument der Funktion ist gleich der benötigten Zahl von Bytes. Diese Anzahl ergibt sich aus der Anzahl der Zeichen des Strings a, die man über den Funktionsaufruf „strlen(a)“ erhält, plus eines weiteren Zeichens für das terminierende Null-Symbol „\0“. Der Rückgabewert der Bibliotheksfunktion `malloc` ist gleich dem Zeiger auf den reservierten Speicherplatz. Die Bibliotheksfunktionsaufruf `strcpy(z, a)` kopiert einen String von `a` nach `z`. Dabei sind `a` und `z` die jeweiligen Anfangspointer der Strings. Rückgabewert ist `z`. Für Strings wird die Ungleichheitsrelation nicht mittels „`!=`“ sondern durch die Bibliotheksfunktion „`strcmp(...)`“ realisiert. Realisierungen der Funktionen `strcpy` und `strcmp` sind im Buch von Kernighan und Ritchie finden.

```
/*LinSearch.C*/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void main(){
 char *x[100], a[80];
 int i, n;

 /*Eingabe*/
 printf("SUCHEN IN EINER FOLGE VON STRINGS\n");
 printf("Eingabe der Strings, bis erstmals ein leerer String eingegeben
                                                                  wird\n");
 n=0;
 do{
  printf("? String = "); gets(a);
  x[n++]=strcpy(malloc(strlen(a) + 1), a);
 } while(a[0]);
 printf("  Anzahl der Strings = %d\n", --n);
 printf("? Suche nach = "); gets(a);

 /*Verarbeitung*/
 x[n]=a; i=0;
 while(strcmp(a, x[i])) i++;

 /*Ausgabe*/
 if(i<n) printf("! Der gesuchte String steht an Position %d", i);
 else printf("Der String kommt in der Folge nicht vor");
}
```

Bei der linearen Suche fallen im Mittel (n+1)/2 Suchschritte an. Die Suche lässt sich beschleunigen, wenn die Elemente des Arrays sortiert sind. Voraussetzung dafür ist, dass für die Elemente des Komponententyps eine *Ordnungsrelation* definiert ist. Um später die Algorithmen allgemein anwenden zu können, müssen wir zunächst klären, was mit Ordnungsrelation gemeint ist.

Definition: Die Relation < ist genau dann eine *vollständige Ordnung* auf der Menge *M*, wenn sie folgende Eigenschaften hat (Knuth, 1973, Stichwort: Ordering):

1. Aus $x<y$ und $y<z$ folgt $x<z$ für beliebige $x, y, z \in M$ (Transitivität).
2. Für alle Paare $x, y \in M$ gilt genau eine der Beziehungen $x<y$, $y<x$ oder $x=y$.

Die Kleiner-Relation für die Zahlen ist eine vollständige Ordnung; ebenso die lexikographische Anordnung von Wörtern. Bei der lexikographischen Anordnung von Wörtern liest man „<„ als „kommt vor“.

Da man sich die Elemente einer vollständig geordneten Menge wie auf einer Linie hintereinander angeordnet denken kann, spricht man auch von einer *linearen Ordnung* (linear ordering) .

Anstelle von „$x < y$ oder $x = y$“ schreibt man „$x \le y$“, in C: „<=„.

Ein Array x, für dessen Komponententyp eine vollständige Ordnung definiert ist, nennen wir *sortiert*, wenn aus i < j folgt, dass x[i] ≤ x[j].

Suchproblem 2: Für ein sortiertes Array

```
KomponentenTyp x[n];
```

und einen vorgegebenen Wert a ist der kleinste Index i gesucht, so dass x[i] = a.

Die Suche lässt sich folgendermaßen organisieren: Wir tasten uns mit dem Index i von unten und mit dem Index j von oben an die gesuchte Stelle heran. Wir wählen den Index i so, dass

unterhalb i nur Elemente kleiner a stehen, und den Index j so, dass a≤x[j] oder j=n gilt. Präziser ausgedrückt, legen wir dem Algorithmus die *Invarianten*

1. (k<i) impliziert (x[k]<a) und
2. (j<n) impliziert (a≤x[j]).

zugrunde. Die Suche bricht ab, wenn i=j. Dann gilt x[i-1]<a≤x[i]. Das heißt: Entweder zeigt i auf die gesuchte Position, oder a kommt im Array überhaupt nicht vor.

Die *Binäre Suche* ist eine Lösung für das Suchproblem 2:

```
i=0; j=n;
 while(i<j){
  int m=(i+j)/2;
  if(strcmp(x[m], a)<0) i=m+1; else j=m;
 }
```

Zur Zeitkomplexität von Algorithmen: Seien *f* und *g* Funktionen der nichtnegativen ganzen Zahlen *n* mit *f*(*n*)=0 und *g*(*n*)=0. Wir sagen, *f*(*n*) ist *höchstens in der Größenordnung* von *g*(*n*), wenn es eine Konstante *k* gibt, so dass ab einem bestimmten Wert *n* gilt: $f(n) \le k \cdot g(n)$. Ist *f*(*n*) höchstens in der Größenordung von *g*(*n*) und umgekehrt *g*(*n*) höchstens in der Größenordnung von *f*(*n*), dann sind beide von derselben Größenordnung.

Beim binären Suchen ist die Anzahl der Schleifendurchläufe in der Größenordnung $\log_2(n)$. Bei der linearen Suche ist der Erwartungswert der Schleifendurchläufe in der Größenordnung n. Zumindest für große n ist die binäre Suche also wesentlich effizienter als die lineare Suche.

Der Algorithmus der binären Suche ist hier - gegenüber dem üblicherweise vorgestellten Algorithmus - leicht modifiziert: Mir scheint er etwas einfacher und klarer zu sein, und darüberhinaus ist er im Ergebnis bestimmter. Er findet nämlich nicht irgendein Element gleich a, sondern genau das erste. Er ähnelt dem aus dem Buch von Robert L. Baber (The Spine of Software. John Wiley 1987, S. 188 ff.). Dort wird allerdings nicht nur das erste, sondern auch noch das letzte Element gleich a identifiziert.

Sortieren von Arrays

Einer der effizientesten Algorithmen für das Sortieren von Arrays ist *Quicksort*. Es handelt sich um eine rekursiven Algorithmus.

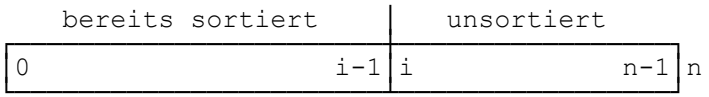
Hier wird ein besonders einfacher und durchsichtiger Algorithmus vorgestellt, nämlich das *Sortieren durch Einfügen* (*Insertion Sorting*). Unter den elementaren Sortieralgorithmen gehört er zu den effizientesten (Wirth, 1983).

Zugrundegelegt wird ein Array gemäß folgender Deklaration

```
KomponentenTyp x[n];
```

wobei für den Komponententyp eine vollständige Ordnung „<“ definiert ist.

Beim *Sortieren durch Einfügen* (Insertion Sort) teilt man das Array in einen bereits sortierten Abschnitt für Indizes < i und den unveränderten Rest auf. Die folgende Skizze gibt einen Überblick.



Diese Bedingung, nämlich dass die Elemente unterhalb von i bereits sortiert sind, ist die wesentliche *Invariante* des Algorithmus. Für i=1 ist sie jedenfalls wahr. Für größere i lässt man den Wert x[i] über schrittweise Vertauschungen (swap) sukzessive nach unten wandern, bis er

an der richtigen Stelle steht. Anschließend wird i um eins erhöht. Die Invariante ist wieder erfüllt. Diese Handlungsanweisung wird solange wiederholt, bis i=n ist. Diese *Endebedingung* zusammen mit der Invarianten liefert das gewünschte Ergebnis: Das Array insgesamt ist sortiert.

Sortieren durch Einfügen (Insertion Sort): Die Komponenten sind hier vom Typ String. Der Aufruf „`strcmp(A, B)`" liefert genau dann einen Wert <0, wenn der String `A` in lexikographischer Ordnung vor dem String `B` steht. Die Variable `des` Programms sind folgendermaßen deklariert:

```
char *x[100], *a;
int i, j, n;
```

Folgende Version des Insertion Sorting ist im Buch von Aho, Hopcroft, Ullman (1983) zu finden:

```
i=1;
while(i<n){
  j=i;
  while(0<j && strcmp(x[j],x[j-1])<0){
    a=x[j]; x[j]=x[j-1]; x[j-1]=a;
    j=j-1;
  }
  i=i+1;
}
```

Übungen

1.6.1 Zeigen Sie für den Algorithmus der binären Suche, dass die Invariante nach jedem Schleifendurchlauf wahr ist. Welche Bedingungen sind nach Beendigung des Programms erfüllt? Ein Ergebnis der binären Suche kann sein, dass i<n und x[i] gleich a gilt. In diesem Fall ist i der gesuchte Index. Andernfalls kommt a im Array überhaupt nicht vor.

1.6.2 Was genau wird im obigen Programmabschnitt von Insertion Sort eigentlich sortiert? Skizzieren Sie, was passiert. Welche Werte weren im Speicher verschoben?

1.6.3 Zeigen Sie, dass der folgende Programmabschnitt ebenfalls das Insertion Sorting realisiert. Warum ist diese Version etwas effizienter als die oben angegebene?

```
for(i=1; i<n; i++) {
  for (a=x[j=i]; 0<j && strcmp(a, x[j-1])<0; j--) x[j]=x[j-1];
  x[j]=a;
}
```

2 Datenstrukturen und Funktionen

2.1 Programmaufbau und Funktionen

Bei der Lösung umfangreicher Aufgaben treten mehrere Probleme auf:

1. Wird die umfangreiche Aufgabe in einem unstrukturierten Gesamtprogramm abgehandelt, geht die Übersicht schnell verloren. Andererseits lassen sich meist Teilaufgaben herauslösen, die miteinander nur recht lose gekoppelt sind und die sich - unter Einhaltung von Schnittstellenvereinbarungen - grundsätzlich separat behandeln ließen. Das gilt beispielsweise für die Teilaufgaben *Eingabe*, *Verarbeitung* und *Ausgabe*.
2. Gewisse Funktionen treten in verschiedenen Zusammenhang immer wieder auf, beispielsweise die Eingabe über Tastatur oder die Berechnung einer mathematischen Funktion. Es ist nicht vorteilhaft, diese immer wieder neu hinzuschreiben. Auch wenn der Editor solche Wiederholungen durch Kopieroperationen leicht macht: die Übersichtlichkeit des Programms leidet darunter. Änderungen und Fehlerbeseitigung werden erschwert.
3. In den verschiedenen Programmierprojekten treten oft ähnliche oder dieselben Aufgaben auf. Durch Übernahme des Programmtexts könnte man von Vorgängerprojekten profitieren. Aber hier entsteht dasselbe Problem wie oben: Änderungen und Verbesserungen in dem einen Projekt haben keine positiven Rückwirkungen auf das andere. Besser wäre es, Bibliotheken von Funktionen zu haben und weiterzuentwickeln.

Für die *Strukturierung* von Programmen (Lösung des Problems 1) und die *Redundanzreduktion* (Lösung der Probleme 2 und 3) bietet C das Konzept der *Funktionen*. Funktionen haben die Aufgabe, Teile eines Programms unter einem Namen zusammenzufassen. Die Eingabedaten einer Funktion werden gegebenenfalls in einer Parameterliste erfasst. Funktionen haben normalerweise einen Rückgabwert. Funktionen ohne Rückgabewert (Typ `void`) nennt man auch Prozeduren.

Nun genügt es überall dort, wo das Teilprogramm erscheinen müsste, den Funktionsnamen und - in Klammern - die aktuellen Parameterwerte anzugeben. Das bewirkt dann jeweils einen *Funktions-* bzw. *Prozeduraufruf*. In den bisherigen Programmen traten u. a. folgende Funktionsaufrufe auf: `scanf(...), printf(...), strlen(x).`

Syntax: Programmaufbau und Funktionen

Ein Programm besteht aus einer oder mehreren zusammenhängenden Dateien. Jede Datei ist eine Übersetzungseinheit *(translation unit)*. Eine Datei besteht aus einer Folge von Funktionsdefinitionen und Deklarationen. Eine Funktion muss den Deklarator `main` haben. Mit der Funktion main startet die Verarbeitung. Die bisher besprochenen Programme kommen allein mit dieser Funktion aus.

```
translation-unit =
  external-declaration { external-declaration }

external-declaration = function-definition | declaration
```

Beispiele für Deklarationen *(declaration)* haben wir bereits kennengelernt. Das soll vorerst genügen. Erst im nächsten Abschnitt werden Deklarationen eingehender behandelt. Dort werden dann auch die benutzerdefinierten Datentypen eingeführt. Hier wollen wir die Funktions-

definition *(function-definition)* genauer betrachten. In etwas vereinfachter Form[1] sieht sie so aus:

```
function-definition =
   { type-specifier } declarator compound-statement

type-specifier = void | char | short | int | long | float |
   double | signed | unsigned | ...
```

Der *Anweisungsteil* einer Funktion ist eine zusammengesetzte Anweisung *(compound-statement)*. Die Typspezifikatoren *(type-specifier)* legen den Typ des Rückgabewertes fest. Der Rückgabewert wird durch die *Return-Anweisung* bestimmt: Sobald der Programmablauf innerhalb der zusammengesetzten Anweisung auf eine Return-Anweisung trifft, wird der Ausdruck hinter dem Schlüsselwort `return` ausgewertet und dieser Wert wird der Rückgabevariablen zugewiesen. Damit endet die Funktion. Return-Anweisungen können innerhalb der zusammengesetzten Anweisung mehrfach auftreten.

Der Deklarator *(declarator)* legt vor allem den Funktionsnamen fest. Im Zusammenhang mit Variablen und benutzerdefinierten Datentypen werden Deklaratoren eingehender behandelt. Bei Funktionen können wir zunächst die folgende - vereinfachte - Form des Deklarators zu Grunde legen:

```
declarator = [ * ] identifier ( [ parameter-list ] )
```

Der Dereferenzierungsoperator * besagt, dass ein Zeigerwert auf eine Variable des spezifizierten Typs zurückgegeben wird. Die *Parameterliste* (*parameter-list*) enthält die Deklarationen der Übergabevariablen (Formalparameter).

Beispiel 1: Die folgende Übersetzungseinheit besteht aus zwei Funktionsdefinitionen. Aufgerufen werden die Funktionen `fabs`, `sqrt`, `printf`, `scanf`, und `betrag`.

```
/*btrg.c*/
#include <stdio.h>
#include <math.h>

float betrag(float x, float y){
  return fabs(x)<fabs(y)? fabs(y)*sqrt(1+(x/y)*(x/y))
         : x==0? 0: fabs(x)*sqrt(1+(y/x)*(y/x));
}

void main() {
  float a, b;
  printf("? Eingabe a b: ");
  scanf("%e%e", &a, &b);
  printf("! Echo: %e  %e\n", a, b);
  printf("! b = %e\n", betrag(a, b));
}
```

Semantik: Die Wirkung von Funktionsaufrufen

Die (partielle) Funktion von Funktionsaufrufen - also ihre Wirkung auf die Variablen - definieren wir mittels eines *Modellcomputers*. Dieser Modellcomputer ist ein um einen Stack erweiterter Von-Neumann-Rechner[2].

---

[1] Insbesondere die Funktionen mit variablen Parameterlisten werden nicht behandelt.

[2] Hinsichtlich der Bezeichnungen und der Ausprägungen des Modells folge ich den Darstellungen von Wirth (1986) und Aho/Sethi/Ullman (1986).

Ihre Wirkung entfalten *Funktionen* aufgrund von Aufrufen innerhalb von Audrücken. Ein solcher Ausdruck kann auch aus einem einzigen Funktionsaufruf bestehen. Bei einer Ausdrucksanweisung, die nur aus einem *Funktionsaufrufe* besteht, spielt der Rückgabewert keine Rolle und nur auf die Nebenwirkungen kommt es an. Der Funktionsaufruf wirkt dann wie ein Prozeduraufruf in der Programmiersprache Pascal.

Im Aufruf steht für jeden Parameter ein Zuweisungsausdruck, dessen Ergebnistyp zum Typ des Parameters passen - also zuweisungsverträglich sein - muss. Beim Aufruf wird für jeden Parameter eine Variable des deklarierten Typs angelegt (aktueller Parameter). Der zugehörige Zuweisungsausdruck wird ausgewertet und das Ergebnis der Variablen zugewiesen. Im Anweisungsteil (Block) der Funktion wird nur auf diese lokale Variable Bezug genommen. Diese Methode der Parameterübergabe heißt *Call by Value*. In C gibt es nur diese Form der Parameterübergabe.

Durch Pointervariablen in der Parameterliste kann man im Funktionsaufruf Zeiger auf Variablen übergeben. Diese Variablen lassen sich dann durch die Funktion verändern. Wir haben das bei den `scanf`-Funktionsaufrufen schon gesehen.

Für jeden Formalparameter der Parameterliste einer Funktion müssen beim Aufruf *aktuelle Parameter* angelegt werden. Auch für den Rückgabewert ist eine Variable vorzusehen. Hinzu kommen eventuell weitere *lokale Variablen*, die innerhalb des Anweisungsteils der Funktion deklariert sind.

Als `static` deklarierte lokale Variablen werden wie externe Variablen für die gesamte Laufzeit des Programms bereitgehalten. Alle anderen lokalen Variablen werden nur benötigt, solange die Funktion abgearbeitet wird. In der Programmiersprache C werden solche Variablen *automatisch* genannt. Erst beim Aufruf einer Funktion wird der für die automatischen Variablen benötigte Speicherplatz reserviert. Nach Beendigung der Funktion wird der Speicher wieder freigegeben.

Funktionsaufrufe können geschachtelt sein: Innerhalb einer Funktion können wiederum Funktionen aufgerufen werden. Aber grundsätzlich sind vor Abschluss einer Funktion alle von ihr aufgerufenen Funktionen abgeschlossen. Wegen dieses Prinzips ist es naheliegend, den Speicherbereich für die automatischen Variablen als *Stack* zu organisieren.

Je Funktionsaufruf wird ein Speicherbereich am oberen Ende des Stacks reserviert. Dieser Speicherbereich nennen wir *Funktionssegment* (auch: Unterprogramm-Segment; englisch: *activation record*).

Die Anzahl der Variablen und deren Speicherbedarf kann je nach Funktion durchaus unterschiedlich sein. Die entsprechenden Speichersegmente können also unterschiedlich lang sein.

Um nun den Stack auch mit Segmenten unterschiedlicher Länge füllen und dennoch diese Segmente sauber auseinanderhalten zu können, wird jedem Segment eine Verwaltungsinformation mitgegeben: Auf einem speziellen Speicherplatz des Unterprogramm-Segments steht die Anfangsadresse des vorhergehenden Unterprogramm-Segments (das ist das Segment des aufrufenden Unterprogramms). Diese Adresse wird *Dynamic Link* (DL) genannt (siehe nebenstehende Tabelle).

Mögliche Struktur eines Funktionssegments

Rückgabevariable
Variable der aktuellen Parameter
Rücksprungadresse RA
Dynamic Link DL
weitere lokale Variable
temporäre Variable (im Zuge der Auswertung von Ausdrücken beispielsweise)

Die DL-Felder sorgen für eine Verkettung sämtlicher Segmente im Stack, angefangen vom obersten bis hinunter zum „Boden" des Kellerspeichers. Dies nennt man die *dynamische Kette* der Segmente.

Mit dem Unterprogrammaufruf wird der Befehlszähler auf die Adresse gesetzt, mit der die Anweisungsfolge des Unterprogramms beginnt. Nach Abarbeitung des Unterprogramms geht es hinter dem Unterprogrammaufruf weiter. Das heißt: Im Unterprogramm-Segment ist auch noch die *Rücksprungadresse* (*Return Address*, RA) aufzubewahren. Die Speicherbereiche für die Verwaltungs- und Steuerungsdaten sind in der obigen Tabelle hervorgehoben.

Die Rückgabevariable eines Funktionssegments kommt als erstes auf den Stack, sie liegt also im Funktionssegment zuunterst. Ganz oben liegen die lokalen und gegebenenfalls die temporären Variablen.

Die Speicheraufteilung und -verwaltung

Während der Laufzeit eines Programmes werden - je nach Speicherinhalt und Art der Speicherverwaltung - verschieden Speicherbereiche unterschieden. Da es hier speziell um den Stack geht, ist er in der folgenden Tabelle hervorgehoben.

Zu den übrigen Speicherbereichen: Der Inhalt des *Programmspeichers* bleibt während der gesamten Laufzeit des Programms unverändert - sowohl in Bezug auf den Umfang als auch auf den Inhalt. Der Speicherbereich für die *statischen Variablen* enthält die extern deklarierten und die speziell als `static` spezifizierten Variablen. Sie sind bereits zur Zeit der Programmübersetzung bekannt und werden für die gesamte Laufzeit des Programms angelegt. Der Umfang dieses Speicherbereich bleibt während der gesamten Laufzeit des Programms unverändert; nur der Inhalt kann sich ändern.

Speicherbereiche

Programm
Statische Variable
Heap
freier Speicher
Stack

Die Speicherbereiche *Stack* und *Heap* sind vom Inhalt und Umfang her variabel. Da sich die Größen dieser Bereiche ändern können, werden sie manchmal an entgegengesetzten Enden des verfügbaren Speicherbereichs untergebracht - mit einer variablen Grenze zum freien Spei-

cherbereich hin[1]. Auf dem Heap werden die dynamischen Variablen untergebracht. Das sind die Variablen, die im Laufe des Programmes beispielsweise mit der `malloc`-Funktion angelegt und mit `free`-Funktion wieder gelöscht werden. Dieser Speicherbereich erfordert gerade wegen der großen Freizügigkeit, eine recht aufwendige Verwaltung[2].

Speicherung von Arrays und Pointern auf Stack und Heap

Zur Erkundung, wie Arrays und Pointer auf Stack und Heap gespeichert werden, dient ein kleines Programm, in dessen main-Funktion folgende Deklarationen stehen:

```
unsigned char *p, c[]="4711", *q, *z=(char*)&q;
```

Der Charakterpointer z soll es erlauben, die einzelnen Zeichen des Pointers q und deren Abfolge im Speicher zu studieren. Außerdem enthält das Programm die Zuweisungen

```
p=q=malloc(5);
*q++='a';
*q++='b';
*q++='c';
*q++='d';
*q=0;
q=c;
```

und die Ausgabeanweisungen:

```
printf("c= %s\n", c);
printf("q= %s\n", q);
printf("p= %s\n", p);
printf("c= %p, *c= %c, &c= %p\n", c, *c, &c);
printf("q= %p, *q= %c, &q= %p\n", q, *q, &q);
printf("p= %p, *p= %c, &p= %p\n", p, *p, &p);
printf("z= %p, *z= %c, &z= %p\n", z, *z, &z);
printf("*z++= %X\n", *z++);
printf("*z++= %X\n", *z++);
printf("*z++= %X\n", *z++);
printf("*z= %X\n", *z);
```

Auf dem Bildschirm erscheint nach Aufruf des Programms diese Ausgabe:

```
c= 4711
q= 4711
p= abcd
c= 0064FDF8, *c= 4, &c= 0064FDF8
q= 0064FDF8, *q= 4, &q= 0064FDF4
p= 00662618, *p= a, &p= 0064FE00
z= 0064FDF4, *z= ø, &z= 0064FDF0
*z++= F8
*z++= FD
*z++= 64
*z= 0
```

---

[1] Aber das ist nicht einheitlich geregelt: Allein Borland C++ unterscheidet sechs Speichermodelle, und nicht alle halten das angegebene Schema ein.

[2] Im Buch von Kernighan und Ritchie (1988) findet man Realisierungen der Funktionen `malloc` und `free` im Abschnitt 8.7 "Example - A Storage Allocator".

Das zeigt, wie die Variablen des Programms auf Stack und Heap untergebracht sind:

Adresse (Stack)	***Inhalt***	***Kommentar***
0064FDF0 (Top of Stack)	F4	Pointer z
0064FDF1	FD	(Achtung: least sig-
0064FDF2	64	nificant byte first)
0064FDF3	00	
0064FDF4	F8	Pointer q
0064FDF5	FD	
0064FDF6	64	
0064FDF7	00	
0064FDF8	'4'	Array c
0064FDF9	'7'	
0064FDFA	'1'	
0064FDFB	'1'	
0064FDFC	'\0'	Fortsetzung Array c
0064FDFD	leer	(terminierende Null)
0064FDFE	leer	
0064FDFF	leer	
0064FE00	18	Pointer p
0064FE01	26	
0064FE02	66	
0064FE03	00	

Stack wächst von den hohen zu den niedrigen Adressen

Adresse (Heap)	***Inhalt***
00662618	'a'
00662619	'b'
0066261A	'c'
0066261B	'd'
0066261C	'\0'
0066261D	leer
0066261E	leer
0066261F	leer

Übung

2.1.1 Machen Sie sich alle Anweisungen und deren Resultate des Übungsprogramms zur Speicherung von Arrays und Pointern klar.

2.1.2 Überlegen Sie sich ein kleines Testprogramm, mit dem Sie die Regeln erkunden können, nach denen der verwendete Compiler Speicherplatz auffüllt (Leerfelder).

2.2 Die Rekursion

Gültigkeitsbereich eines Funktion erstreckt sich vom Punkt, wo ihr Name er deklariert wurde bis ans Ende der Compilierungseinheit (Datei). Deshalb ist es durchaus zulässig, eine Funktion bereits in dem Anweisungsteil der Funktion selbst aufzurufen. Funktionen, die sich selbst aufrufen, heißen *rekursiv*. Das kann auch auf Umwegen über andere Funktionen, also indirekt, passieren.

Einführendes Beispiel: Textinversion

Was liefert das folgende Programm[1], wenn man den String 'Ein Neger mit Gazelle zagt im Regen nie' eingibt?

```
/*Texti.c*/
#include <stdio.h>
ReadWrite(){
 char x = getchar();
 if (x!='\n') {ReadWrite(); putchar(x);}
}

main(){
 printf("\nTEXTINVERSION (Eingabeende: CR)\n");
 ReadWrite();
}
```

Schritt für Schritt lässt sich mittels „Papiercomputer“ nachvollziehen, was das Programm Texti.c tut. Die folgende Tabelle stellt die Speicherbereiche zusammen. Die variablen Informationen stehen im Arbeitsspeicher, in den peripheren Speichern und in Registern. Eingabemedium ist die Tastatur und im Feld Eingabepuffer steht der Inhalt des Tastaturpuffers. Die Anweisungen im Programmspeicher sind durchnummeriert. Auf diese Nummern bezieht sich der Befehlszähler (PC, program counter).

Der Programmablauf startet mit dem ersten Befehl. Der Stack ist leer, denn es ist noch keine Funktion aufgerufen worden und das Programm hat keine statischen Variablen. Es ist noch nichts eingegeben worden und der Cursor des Ausgabebildschirms (dargestellt als senkrechter Strich) steht auf der ersten Position. Diese Situation ist in der Tabelle dargestellt.

Der weitere Programmablauf ist als Animationsfolge auf der Informatik-Seite im Web zu finden. Adresse: `http://www.fh-fulda.de/~grams/informat.htm`.

Programmspeicher:	**PC:** 1

`ReadWrite(){`
` char x = `[3]`getchar();`
` `[4]`if (x!='\n') {`
`  `[5]`ReadWrite();`
`  `[6]`putchar(x);`
` }`
`}`
`main(){`
` `[1]`printf("\nTEXTINVERSION ...");`
` `[2]`ReadWrite();`
`}`
	Stack:
	
	Eingabepuffer:
	
	Ausgabe:
	|

[1] Die Anregung, die Textinversion als einführendes Beispiel für die Rekursion zu nehmen, habe ich der Overflow-Kolumne des Informatik-Spektrums entnommen (Nievergelt, J.: Schulbeispiele zur Rekursion. Informatik-Spektrum 13 (1990), 106-108).

Die Türme von Hanoi

Die „Türme von Hanoi“ sind *das* Informatik-Puzzle schlechthin. Es darf in keiner einführenden Darstellung in die Informatik fehlen. Tatsächlich ist es eines der eindrucksvollsten Beispiele für die Rekursion: Der rekursive Ansatz macht hier aus einer zunächst entmutigend schwer erscheinenden Aufgabe ein Kinderspiel.

Die Türme von Hanoi: Gegeben seien drei Pflöcke, die wir mit A, B und C bezeichnen, **Bild 2.2-1**. Zu Beginn steckt auf Pflock A eine gewisse Anzahl von Scheiben. Die Scheiben werden von unten nach oben immer kleiner. Folgende Grundregel ist während des ganzen Spiels einzuhalten: Scheiben werden grundsätzlich auf Pflöcke gesteckt und nie daneben gelegt, und außerdem darf nie eine Scheibe auf einer kleineren liegen. Die Aufgabe lautet, die Scheiben - unter Beachtung der Regel - von Pflock A nach Pflock B zu transportieren.

Gesucht ist ein Programm, das für eine beliebig eingebbare Anzahl *n* von Scheiben das Problem der Türme von Hanoi löst. Auf dem Bildschirm soll die Folge der Verschiebungen erscheinen, z.B.: A->B, A->C, B->C usw.

Lösungsstrategie: Man führt das Problem der *n* Scheiben auf eins von *n*-1 Scheiben zurück. Wenn das Problem für *n*-1 Scheiben gelöst ist, kann man ja die oberen *n*-1 Scheiben nach C bringen. Die unterste Scheibe kommt dann nach B. Schließlich transportiert man die *n*-1 Scheiben von C nach B. Für die Lösung des Problems mit *n*-1 Scheiben wiederholt sich dieser Vorgang. Mittels einer rekursiven Funktion ergibt sich eine elegante Lösung der Aufgabe.

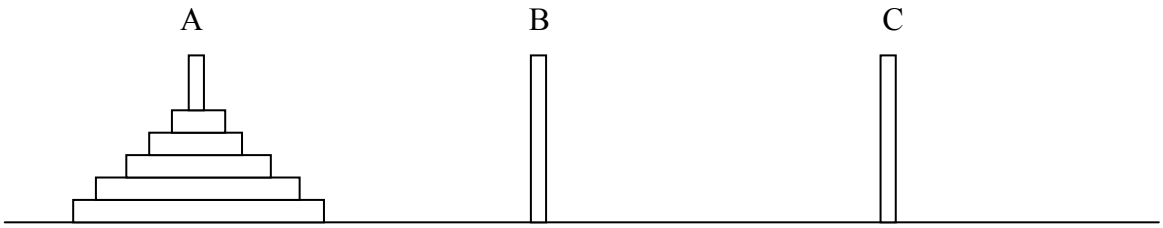


Bild 2.2-1 Die Türme von Hanoi

Eine Lösung in C:

```
/*Hanoi.c*/
#include <stdio.h>

/*A, B, C: Quell-, Ziel und Hilfsturm*/
void move(char A, char B, char C, int n){
 if(n>1)  move(A, C, B, n-1);
 printf("%5d:%c ---> %c\n", n, A, B);
 if (n>1) move(C, B, A, n-1);
}

void main(){
 int n;
 printf("DIE TšRME VON HANOI\n");
 printf("? Anzahl der Scheiben = "); scanf("%d", &n);
 move('A', 'B', 'C', n);
}
```

Zahlenwandlung rekursiv

Das folgende Programm zur Umwandlung einer Dezimalzahl in eine Darstellung mit der Basis b (1<b<10) nutzt ebenfalls die Rekursion.

```
int z, b;

void EvalWrite(int z) {
  if (b<=z) EvalWrite(z/b);
  printf("%d", z%b);
}

main() {
  printf("\nERIKS TRICK\n? z = "); scanf("%d", &z);
  printf("? b = "); scanf("%d", &b);
  printf("! ");
  EvalWrite(z);
}
```

Übungen

2.2.1 Probieren Sie das Spiel „Türme von Hanoi“ mit fünf Münzen aus.

2.2.2 Wie groß ist die Anzahl *f*(*n*) der insgesamt erforderlichen Transporte einzelner Scheiben beim Programm „Türme von Hanoi“? (Mit *n* wird die Gesamtzahl der zu verschiebenden Scheiben bezeichnet.)

2.2.3 Gehen Sie das Programm für Zahlenwandlung am Beispiel z = 202 und b = 5 Schritt für Schritt mittels Papiercomputer durch.

2.2.4 Erläutern Sie die Wirkung des folgenden einzeiligen C-Programms.

```
main(){char* c="main() {char* c=%c%s%c; printf(c, 34, c, 34, 10);}%c";
                                                printf(c, 34, c, 34, 10);}
```

Das Beispiel ist aus dem Aufsatz "The Tragedy of Programming Language Development" von J. Gutknecht aus der Zeitschrift Stuctured Programming (1993) 14, 49-55.

2.3 Benutzerdefinierte und rekursive Datentypen

Bei den bisher eingeführten Datentypen hat der Programmierer keinen oder nur geringen Einfluss auf dessen Ausprägung. Einzig der Array-Typ war in bescheidenem Maße gestaltbar durch die vom Anwender vorgebbare Länge des Arrays. Moderne Programmiersprachen eröffnen einen viel größeren Gestaltungsspielraum. Bereits in der Frühzeit der Programmierung wurde der *Datensatz* oder *Verbund* eingeführt. Das ist eine Zusammenfassung von mehreren *Datenfeldern* mit gleichen oder auch verschiedenen Datentypen zu einem *neuen Datentyp*. In Pascal ist das der *Record-Typ* und in C ist es der *Struktur*-Typ.

Deklaration einer Struktur (Beispiel)

Bevor wir zur allgemeinen Syntax der Deklarationen kommen, wollen wir am Beispiel einer Struktur den Aufbau einer Deklaration studieren:

```
struct complex {float x, y;} a[3]= {{3, 4}, {-6, 8}, {9, -12}};
```

Die Deklaration ist mit einem Semikolon abgeschlossen und enthält die *Deklarationsspezifikatoren (declaration-specifiers)* - hier nur den *Strukturspezifikator (struct-specifier):*

```
struct complex {float x, y;} a[3]= {{3, 4}, {-6, 8}, {9, -12}};
```

einen *Deklarator (declarator)*

```
struct complex {float x, y;} a[3]= {{3, 4}, {-6, 8}, {9, -12}};
```

und einen *Initialisator (initializer)*

```
struct complex {float x, y;} a[3]= {{3, 4}, {-6, 8}, {9, -12}};
```

Im vorliegenden Fall leistet die Deklaration folgendes.

1) Für die Darstellung komplexer Zahlen wird der neue Datentyp `struct complex` als Struktur mit den Feldern `x` und `y` spezifiziert.
2) Die Variable `a` wird als Array dreier Elemente dieses neuen Typs deklariert und definiert.
3) Die Felder der Variablen a werden mit den komplexen Zahlen 3+j4, -6+j8 und 9-j12 initialisiert

Eine Deklaration muss nicht alle diese Komponenten enthalten. Die Variablen einer Struktur - im Beispiel die Variablen `x` und `y` - werden im C-Sprachgebrauch auch *Member* genannt. Der Zugriff auf Member geschieht mittels Punkt-Notation: Die Member-Variable `a[1].x` hat beispielsweise den Wert -6.

Syntax der Deklarationen (Selbststudium)

In der Syntaxdefinition der Deklarationen sind nicht enthalten die Typen Enumeration `(enum)` und Union `(union)`.

declaration = *declaration-specifiers* [*init-declarator-list*] **;**

init-declarator-list = *init-declarator* { **,** *init-declarator* }

init-declarator = *declarator* [**=** *initializer*]

```
initializer =
  assignment-expression | { initializer { , initializer } }

declaration-specifiers = { declaration-specifier }

declaration-specifier = storage-class-specifier |
  type-specifier | const

storage-class-specifier = static | extern | typedef | ...

type-specifier = void | char | short | int | long | float |
  double | signed | unsigned | struct-specifier | identifier

struct-specifier =
  struct [ identifier ] { struct-declaration-list } |
  struct identifier

struct-declaration-list =
  struct-declaration { struct-declaration }

struct-declaration =
  specifier-qualifier-list struct-declarator-list ;

specifier-qualifier-list = [ const ] { type-specifier }

struct-declarator-list = declarator { , declarator }

declarator = [ pointer ] direct-declarator

direct-declarator = identifier | ( declarator ) |
  direct-declarator [ [ constant-expression ] ] |
  direct-declarator ( [ parameter-list ] )

pointer = * [ const ] [ pointer ]
```

Ein konstanter Ausdruck (constant-expression) ist ein Ausdruck ohne Nebenwirkungen und ohne Funktionsaufrufe.

Für die Typumwandlung, die Speicherplatzreservierung und in Parameterdeklarationen werden *Typ-Namen* benötigt. Ein Typname *(type-name)* ist im Wesentlichen eine Deklaration, in dem der Variablenname *(identifier)* und - und damit auch der Initialisator - fehlt.

```
type-name = specifier-qualifier-list [ abstract-declarator ]

abstract-declarator =
  pointer | [ pointer ] direct-abstract-declarator

direct-abstract-declarator = ( abstract-declarator ) |
  [ direct-abstract-declarator ] [ [ constant-expression ] ]
  [ direct-abstract-declarator ] ( [ parameter-list ] )

parameter-list =
  parameter-declaration { , parameter-declaration }

parameter-declaration = declaration-specifiers declarator |
  declaration-specifiers [ abstract-declarator ]
```

Zusammenfassung: Die Deklarationsspezifikatoren legen einen Datentyp fest. Das geschieht über die Angabe des Namens für den Datentyp oder die Definition eines neuen Datentyps, für den normalerweise gleich ein Name festgelegt wird. Dieser Name kann in weiteren Deklarationen verwendet werden. Die Deklaratoren stellen Namen für *Objekte* (Variablen) vom spezi-

fizierten Typ zur Verfügung und reservieren den Speicherplatz für das Objekt. Gleichzeitig kann die Variable mit bestimmten Werten initialisiert werden. Abstrakt-Deklaratoren sind Deklaratoren ohne Variablennamen.

Neue Datentypen werden mit typedef definiert. Eine solche Definition sieht aus wie eine Variablendeklaration, mit dem Unterschied, dass der deklarierte Name nicht für eine Variable, sondern für den Datentyp steht. Beispiel:

```
typedef struct {float x, y;} compl;
compl z = {2.3, 1.7};
```

Hier wird der Datentyp `compl` für komplexe Zahlen definiert. Anschließend erscheint die Deklaration einer komplexen Zahl, deren Realteil auf 2.3 und deren Imaginärteil auf 1.7 initialisiert wird.

Anwendung: Speicherung einer Bahnkurve

Nehmen wir einmal an, wir wollten die Bahn eines Roboters in der Ebene mit einer Punktfolge beschreiben: (0, 0), (0.125, 0.08), (0.25, 0.2), ..., (-1.77, 4.65). Im Rechner könnte man die Punktfolge durch ein Array darstellen, das folgendermaßen deklariert ist:

```
struct complex Bahn[1000];
```

Wir nehmen an, dass die die Bahnkurve erst während der Laufzeit des Programms entsteht. Es ist nicht von vornherein klar, wie viele Punkte abzuspeichern sind. Die Zahl 1000 ist mehr oder weniger willkürlich gewählt in der Hoffnung, dass die Zahl der Felder des Arrays ausreicht. Offensichtlich kann diese Art der Speicherreservierung sehr unwirtschaftlich sein. Etwas besser ist es, die Variablen zur Laufzeit dynamisch anzulegen und das Array als Array von Zeigern zu definieren:

```
struct complex *Bahn[1000];
```

Jetzt ist der von vornherein vorzusehende Speicherplatz deutlich geringer, denn jedes Array-Feld nimmt nur noch den Platz für einen Zeiger ein. Jetzt kann der Umfang des Arrays großzügiger festgelegt werden. Allerdings belegen die nun während der Laufzeit anzulegenden Punkte vom Typ `struct complex` zusätzlichen Speicher auf dem Heap. Grundsätzlich würde man das Array mit Null-Pointern vorbelegen:

```
for (i=0; i<1000; i++) Bahn[i]=NULL;
```

Und das Anlegen eines Punktes geschieht mittels `malloc`, z. B.:

```
Bahn[2]=malloc(sizeof(stuct complex));
Bahn[2]->x=0.25; Bahn[2]->y=0.2;
```

Anmerkung: Wir haben bisher die Funktion `malloc` nicht normgerecht verwendet. Der Typ des Rückgabewertes von `malloc` passt nämlich nicht zum Typ der Variablen `Bahn[0]`. Eine explizite Typumwandlung (type cast) ist angezeigt. Ergänzen Sie die obige Anweisung demgemäß.

Eine dynamische Variable, auf die ein Zeiger p verweist, erhält man durch *Dereferenzierung,* in Zeichen: *p. Falls *p eine Struktur ist, kann man auf eines seiner Member m mittels (*p).m oder mittels p->m zugreifen. Das heißt: die obige Zuweisung „`Bahn[2]->x=0.25;`“ ist gleichbedeutend mit „`(*Bahn[2]).x=0.25;`“.

Datenmüll und baumelnde Zeiger

Ein Zeiger ist selbst eine Variable, die ihren Wert durch Zuweisung ändern kann. Auf diese Weise können Referenzen verloren gehen, wenn man den ursprünglichen Wert nicht anderswo aufgehoben hat. Dann entsteht der berüchtigte Datenmüll *(Garbage)*.

Der Speicherplatz dynamischer Variablen kann durch die Standardfunktion free wieder freigegeben werden. Der Aufruf `free(p)` löscht die dynamische Variable `*p`.

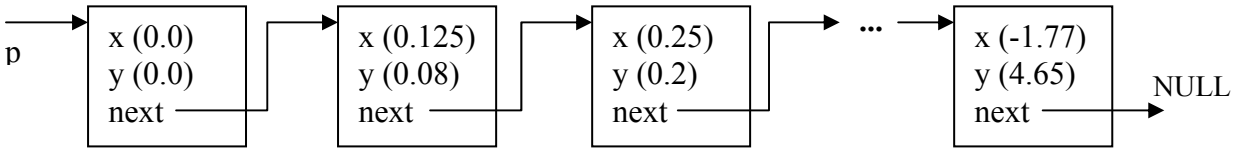
Auch das hat seine problematische Seite: So wie einer dynamischen Variablen der Zeiger abhanden kommen kann, so verliert mittels free der Zeiger seine Variable. Viele Programmierfehler sind auf diese sogenannten *baumelnden Zeiger (Dangling Reference)* zurückzuführen.

Rekursive Datenstrukturen

Die obige Array-Lösung für die Bahnkurve ist nicht das Nonplusultra. Schöner wäre es, wenn man auf ein Array für die Referenzen verzichten könnte. Hier helfen *rekursive (oder selbstreferenzierende) Datenstrukturen* weiter: Es ist erlaubt, innerhalb einer neu definierten Datenstruktur ein Member vorzusehen, dessen Typ ein Pointer auf ein Element gerade dieses Typs ist. Wir definieren unter diesem Gesichtspunkt die Struktur für komplexe Zahlen neu. Dabei machen wir uns über eine `define`-Direktive gleichzeitig von lästigen Wiederholungen des Schlüsselworts struct frei:

```
#define complex struct complex
complex {float x, y; complex *next;};
```

Tip: Schauen Sie sich bei dieser Gelegenheit einmal an, welche Möglichkeiten der Präprozessor mit den `include`- und `define`-Direktiven bietet. Unter anderem kann man mnemotechnisch vorteilhafte Kurzzeichen für Konstante und Datentypen einführen. Auch kann man Makros mit Parametern definieren, die eine effiziente Alternative zur Funktionsdefinition sein können.



Jetzt können wir die Bahnkurve als *lineare Liste* abspeichern:

Das folgende Programmfragment zeigt, wie man eine solche Liste durch direkte Eingabe über die Tastatur aufbauen kann:

```
complex *p=0, *h;
unsigned char c='j';
printf("\nListe komplexer Zahlen. Eingabe:\n");
while (c=='j') {
  h=(complex*)malloc(sizeof(complex));
  printf("? x y= "); scanf("%f %f", &h->x, &h->y);
  h->next=p;
  p=h;
  printf("? Weiter mit \"j\": ");
  scanf("%1s", &c);
}
```

Es wird nur für die Punkte Speicherplatz reserviert, die tatsächlich auch eingegeben worden sind. Die obige Skizze ist etwas irreführend, als sie suggeriert, dass die Speicherplätze für die komplexen Zahlen fein säuberlich hintereinander im Speicher stehen. Das muss keineswegs der Fall sein: Jedes dieser „Kästchen“ kann irgendwo auf dem Heap „liegen“.

Beispielprogramm: Assoziativarray (Selbststudium)

Das folgende Beispielprogramm nimmt Wörter (Strings) von der Tastatur entgegen. Schließlich gibt es eine Tabelle dieser Wörter ohne Wiederholungen aus. In einer zweite Spalte steht, wie oft das jeweilige Wort eingegeben worden ist.

```
/*Assoziativarray
Variante des Programms aus Stroustrup, 1995, Abschnitt 2.3.10 "Refer-
ences"*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define large 1024

struct pair{char* name; int val;};

struct pair vec[large+1];                /*Die externe Tabelle wird zu null
                                                        initialisiert*/

/*Verwaltung eine Menge von Paaren. Sucht nach p und liefert den Zeiger auf
das entsprechende Paar. Falls p noch nicht vorkommt, wird der Zeiger auf
ein noch unbenutzes Paar geliefert.*/
struct pair* find(const char*p){
 int i;
 for (i=0; vec[i].name; i++) if (strcmp(p, vec[i].name)==0) return &vec[i];
 if (i==large) return &vec[large-1];
 return &vec[i];
}

int* value(const char* p) {
 struct pair* res = find(p);
 if (res->name==0) {                    /*Initialisieren eines neuen Paares*/
  res->name = malloc(strlen(p) + 1);              /*Speicherreservierung*/
  strcpy(res->name, p);
 }
 return &res->val;
}

void main() {
 char buf[256];
 int i;
 printf("Haeufigkeit eingegebener Woerter (Abschliessen mit \".\")\n");

 scanf("%s", buf);
 while (buf[0]!='.'){(*value(buf))++; scanf("%s", buf);}

 printf("\n");
 for (i=0; vec[i].name; i++) printf("%s: %u\n", vec[i].name, vec[i].val);
}
```

Übungen

2.3.1 Zeichnen Sie das komplette Syntaxdiagramm für das Demonstrationsbeispiel „Deklaration einer Struktur“.

2.3.2 Was bedeuten die folgenden Deklarationen? Wie sehen die Syntaxbäume aus?

```
int* v[10];
int (*p)[10];
char **cpp;
int (*fp)(char, char*);
struct compl {float re; float im;};
void (*drawLine[6])()={drawLine0, drawLine1, drawLine2,
```

```
                        drawLine3, drawLine4, drawLine5};
```

2.3.3 Zeichnen Sie das Syntaxdiagramm zu Deklaration

```
char z[2][3] = {{'a', 'b', 'c'}, {'0', '1', '2'}};
```

2.3.4 Was leistet der Ausdruck `(*value(buf))++` im Programm „Assoziativspeicher“. Warum darf man die äußeren Klammern nicht weglassen? Was passiert bei der Speicherreservierung mittels `malloc`?

2.4 Funktionen und Module

Größere Programme sind meist auf mehrere separat übersetzbare C-Dateien aufgeteilt. In jeder dieser Dateien müssen zumindest die Deklarationen der in ihr verwendeten Datentypen, Variablen und Funktionen vorhanden sein. Andererseits dürfen Definitionen, das sind Funktionsdefinitionen oder Deklarationen, die für eine Variable Speicherplatz reservieren, nur einmal vorkommen. Sonstige Deklarationen dürfen mehrfach erscheinen.

Deklarationen mit `extern`-Spezifikator und ohne Initialisator bzw. Funktionsrumpf sind *keine* Definitionen. Sie bewirken keine Speicherplatzbelegung. Deklarationen dürfen im Gültigkeitsbereich der Bezeichner mehrfach erscheinen - nicht hingegen Definitionen (Kernighan/Ritchie, S. 227). In einem C-Programm muss es für jeden Namen genau eine *Definition* geben. Das Schlüsselwort `extern` veweist darauf, dass die Definition anderswo steht.

Kopierbare Deklarationen, die in mehreren C-Dateien benötigt werden, fasst man zu sogenannten Header-Dateien zusammen. Diese werden mit dem Include-Mechanismus in die verschiedenen C-Dateien eingebunden. Dadurch wird die Gleichheit der Deklarationen gewährleistet.

Ein Modulkonzept für C

Das folgende Modulkonzept lässt sich mit C realisieren. Es handelt sich aber nicht um ein Sprachelement von C. Statt Interface-Datei sagt man in C auch Header-Datei.

Modul = Interface-Datei + Implementation-Datei

kurz: Modul *name* = *name*.h + *name*.c

oder: Modul *name* = *name*.h + *name*.obj

In eine Header-Datei kommen alle Deklarationen, die exportiert werden sollen. Header-Dateien dürfen Deklarationen enthalten, die keine Definitionen sind und außerdem Typdefinitionen. Da der Anwender eines Moduls seine Informationen über das Modul aus der Headerdatei bezieht, sind darin auch die Kommentare zu den Funktionen und Datentypen enthalten. enthalten außerdem die Spezifikationen der Funktionen und Kommentare zu den Datentypen.

Beispiel:

Interface-Datei:

```
/*Pow.H*/
extern float pow(float x, int n);
/*precondition: n>=0
postcondition: pow = x^n*/
```

Implementation-Datei:

```
/*Pow.c*/
#include "pow.h"
float pow(float x, int n)
{ if (n==0) return 1; else return x*pow(x, n-1); }
```

Hauptprogramm:

```
/*Zweierpo.C*/
#include <stdio.h>
#include "pow.h"
main()
{
```

```
  int i=0;
  printf("Zweierpotenzen\n");
  while (i<10) {
    printf("2 pow %2d = %3.0f\n", i, pow(2, i));
    i++;
  }
}
```

Durch die Präprozessor-Direktive #include wird der Text der bezeichneten Datei eingefügt. Header-Dateien werden mit include-Direktiven textgetreu in die über diese Schnittstelle miteinander kommunizierenden Dateien übernommen, insbesondere auch in die C-Datei des Moduls selbst. Dieser Mechanismus stellt sicher, dass die Typ-Deklarationen eines Namens in den verschiedenen Dateien auch tatsächlich übereinstimmen. Module können separat compiliert werden.

Gültigkeitsbereiche von Deklarationen

- Der Gültigkeitsbereich einer externen Variablen oder Funktion (extern = nicht lokal zu einer Funktion) ersteckt sich ab Deklaration bis Dateiende.
- Steht die Definition einer Variablen in einer anderen Datei, ist eine Extern-Deklaration nötig (mit Speicherklassenspezifikator extern).
- Sowohl externe (also nicht lokale) sowie als statisch deklarierte Variablen (Speicherklassenspezifikator static) werden zu 0 initialisiert, wenn eine explizite Initialisierung fehlt.
- Automatische (also lokale und nicht als statisch deklarierte) Variablen, die nicht explizit initialisiert werden, haben undefinierte Anfangswerte.
- Die Static-Deklaration beschränkt den Gültigkeitsbereich eines Objekts (einer Variablen) auf die Datei.

Header-Dateien

Das kann in Header-Dateien stehen:

- Typ-Definitionen wie struct point {int x, y;}
- Funktionsdeklarationen wie extern float f(float x);
- Variablen-Deklarationen wie extern int a;
- Konstantendefinitionen wie const float pi = 3.1415...
- Include Direktiven wie #include „pow.h“
- Kommentare wie /*Das ist ein Kommentar*/

Tip: Interface-Dateien nicht compilieren; sonst kommt es bei der hier vorgeschlagenen Namenskonvention zu Namenskollisionen!

Zeiger auf Funktionen

Demoprogramm (Kernighan/Ritchie, S. 119):

```
/*FnctPntr.C, Demonstration: "Zeiger auf Funktion"*/

#include <stdio.h>
#include <math.h>

void write(double (*function)(double)) {
  double x;
  for (x=0; x<5.1; x+=.5) printf("\nx=%lf --> y=%lf", x, function(x));
}

main() {
  printf("\nExpontentialfunktion:"); write(exp);
  printf("\nSinusfunktion:"); write(sin);
}
```

Der Funktionsname steht auch für den Pointer auf die Funktion. Das ist analog zur Konvention bei den Arrays.

Übersetzen, Binden, Starten von Programmsystemen

Mit der Modularisierung kommen wir vom Programmieren im Kleinen zum Programmieren im Großen, zum *Software-Engineering*.

Die Bewältigung großer Aufgaben braucht neue Techniken - die Modularisierung ist eine davon - und neue Methoden.

Fassen wir die Gründe für die Modularisierung[1] zusammen:

- Große Aufgaben können in bewältigbare kleinere Aufgaben zerlegt und somit gelöst werden. Die Aufteilung der Teilaufgaben auf verschiedene Arbeitsgänge und -gruppen wird möglich.
- Strukturierungsgrundsätze wie das *EVA-Prinzip* (Eingabe, Verarbeitung, Ausgabe), insbesondere die Trennung von Verarbeitung und Darstellung können wirksam werden.
- Durch separat compilierbare (übersetzbare) Module lässt sich *Zeit sparen*: Was fertig ist, muss nicht immer wieder neu übersetzt werden
- Vorübersetzte Module lassen sich als *Fertigprodukte* weiterverwerten, ohne dass derjenige, der solche Module in sein Programm integriert, die innerer Struktur und den Quelltext es Programms selbst kennt (Information Hiding). Das dient der
- *Entkopplung der Entwicklung* und damit der Verringerung der Fehleranfälligkeit in der Entwicklung sowie
- der *Verantwortungsabgrenzung*. Außerdem ist auf diese Weise ein gewisser
- *Urheberrechtsschutz* möglich.

Das kann natürlich nur funktionieren, wenn sich die vorübersetzten Programm tatsächlich zu einem Ganzen verbinden lassen.

Aus diesem Grund ist der Übersetzungsprozess vom Quellcode zum ablauffähigen Programm zweistufig: Zuerst kommt der eigentliche Compiler. Er sorgt vor allem für die Übersetzung

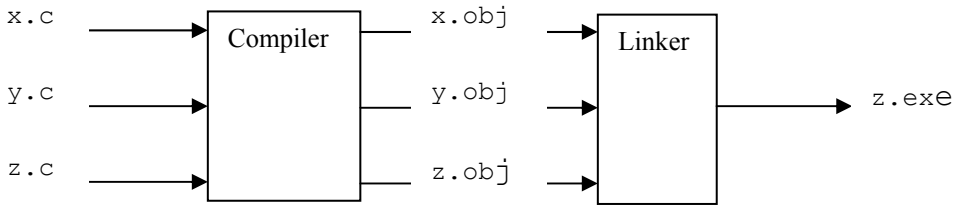
---

[1] Blieberger, J.; Schildt, G.-H.; Schmid, U.; Stöckler, S.: Informatik. Springer, Wien 1990, S. 246

der Module. Die vorübersetzten Module heißen Objektdateien. Sie haben die Dateiendung `.obj`.

Der Compiler compiliert - falls notwendig - die Dateien und startet den Linker. Er ruft den Linker aber nicht nur mit diesen Dateien sondern auch noch mit den nötigen (vorübersetzten) Bibliotheksdateien auf.[2]

Bei unseren bisherigen Programmen haben wir von dieser Zweistufigkeit nichts bemerkt. Es kann uns auch weiterhin ziemlich egal bleiben, denn man kann den Compiler wie bisher aufrufen, nur dass man jetzt mehrere Programme - ob vorübersetzt oder als Quelltext - hinter dem Compileraufruf (durch Leerzeichen voneinander getrennt) angeben muss.



Der Compileraufruf kann jetzt recht umfangreich werden. Das ist mühsam, wenn man ein Programmstück immer wieder übersetzen muss und dazu all die anderen - bereits vorübersetzten Module - mit aufführen muss. Zur Arbeitserleichterung kann man sich eine Stapeldatei (Endung: `.bat`) für die Compilierung anlegen. Diese kann dann die Aufrufe besorgen.

Eine solche Stapeldatei ist die Vorform sogenannter MAKE-Dateien. Diese automatisieren die Projektverwaltung noch weitergehend, indem sie beispielsweise die Notwendigkeit einer Neucompilierung eines Moduls anhand des Erstellungsdatums feststellen (Darnell/Margolis, S. 395-398) und sie dann selbsttätig in die Wege leiten.

Der nächste Schritt ist dann der Übergang auf eine integrierte Entwicklungsumgebung wie Borland C++, die über eine grafische Benutzeroberfläche den Zugang zu den benötigten Werkzeugen (Compiler, Linker, Debugger) sowie die Verwaltung von Projektdateien (MAKE-Files, Anpassung der Bedienoberfläche) erleichtert.

Tip 1: Für jedes Projekt sollte legt man am besten ein eigenes Verzeichnis (Ordner) an. Sie können dann im jeden der Ordner den Übersetzungsaufruf mit der Datei make.bat besorgen.

Tip 2: Am besten öffnen Sie für die Arbeit gleichzeitig zwei DOS-Fenster: Eines für den Editor und das andere für die Compilierung und Fehlermeldungen.

Ein-/Ausgabe von Textdateien

Textdateien bieten eine einfache Möglichkeit, Ergebnisse zwischen Programmen auszutauschen. So kann man beispielsweise die Ergebnisse eines C-Programms mittels Tabellenkalkulation grafisch darstellen.

Umgekehrt kann es auch sinnvoll sein, Daten mit einem Tabellenkalkulationsprogramm zu erfassen und vorzuverarbeiten. Diese Tabelle kann dann als Input in ein C-Programm dienen, mit dem dann die anspruchsvolleren Bearbeitungsschritte erfolgen.

---

[2] Borland C++, Version 5. Benutzerhandbuch. Borland International 1996, S. 227. Zu den Compileroptionen: Quick Tour & Quick Reference, S. 45 ff.

Diese Technik ist geeignet für kleinere Alltagsanwendungen, in denen man rasch vorzeigbare Ergebnissen braucht und bei denen sich eine eigene Programmentwicklung nicht lohnt. Wie so etwas einfach zu realisieren ist, zeigen die beiden folgenden Demonstrationsprogramme:

```
/*TextOut.c
Borland Turbo C++ Einfuehrung, S. 123 f.*/

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

FILE *textfile;
char line[81];

void main() {
  printf("Schreiben der Textdatei:\n");
  if ((textfile=fopen("TEXTIO.TXT", "w"))==NULL) {
    printf("Fehler beim Oeffnen zum Schreiben"); exit(0);
  }
  fprintf(textfile, "%s\n", "eins");
  fprintf(textfile, "%s\n", "zwei");
  fprintf(textfile, "%s\n", "drei");
  fclose(textfile);
}

/*TextIn.c
Borland Turbo C++ Einfuehrung, S. 123 f.*/

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

FILE *textfile;
char line[81];

void main() {
  printf("Lesen der Textdatei:\n");
  if ((textfile=fopen("TEXTIO.TXT", "r"))==NULL) {
    printf("Fehler beim Oeffnen zum Lesen"); exit(0);
  }
  while((fscanf(textfile, "%s", line)!=EOF))
     printf("%s\n", line);
  fclose(textfile);
}
```

Übung

2.4.1 Schreiben Sie Funktionen zum Sichern und Lesen von Textdateien mit angefügter *Checksum*. Die Lesefunktion soll eine Überprüfung der Checksum durchführen und das Ergebnis mitteilen.

Checksum ist ein einfaches Verfahren zur Fehlererkennung. Die Checksum ist die numerische Summe über die Ordinalzahlen aller Zeichen der Datei (modulo UCHAR_MAX+1). Dieser Wert wird als letztes Zeichen (Typ: char) der Datei hinzugefügt.

2.5 Programmierstudie: Datenkompression

Entropie. Datenkompression mittels Huffman-Code.

Diese und die folgende Lektion runden den Kurs „Einführung in die Informatik“ insofern ab, als hier Konzepte und Fragestellungen aus früheren Lektionen aufgegriffen werden: Es soll ein Problem der binären Codierung mittels Programm gelöst werden. Die Lösung nutzt die höheren Konzepte der Programmierung: rekursive Funktionen und rekursive Datenstrukturen (Lektionen 2.4).

Datenkompression

In der Lektion 1.2 haben wir uns mit der binären Codierung von Nachrichten befasst. Dabei ging es nur um Blockcodes: Die Anzahl der Bits je Codezeichen ist bei den Blockcodes konstant. Das ist nicht immer wirtschaftlich. Oft kommt es darauf an, Nachrichten mit möglichst wenigen Bits darzustellen.

Es bietet sich an, die häufiger auftretenden Zeichen zu Lasten der selteneren mit weniger Bits zu codieren. Nehmen wir einmal einen Code für die vier ursprünglichen Zeichen a, b, c und d. Diese mögen mit unterschiedlichen Wahrscheinlichkeiten auftreten. **Tabelle 2.5-1** zeigt für diesen Fall zwei Codes.

Tabelle 2.5-1 Zwei Codes für den Zeichenvorrat {a, b, c, d}

Zeichen	Wahrscheinlichkeit	Code 1	Code 2
a	1/2	00	0
b	1/4	01	10
c	1/8	10	110
d	1/8	11	111

Wie messen den Codierungsaufwand an der mittleren Codewortlänge: Die *mittlere Codewortlänge* (statt Codewortlänge auch: *Elementarzeichenbedarf*) ist gleich der mittleren Anzahl von Bits je ursprünglichem Zeichen. Dabei wird die Wahrscheinlichkeit des Auftretens der Zeichen berücksichtigt. Es handelt sich bei diesem Mittelwert also um den *Erwartungswert* im Sinne der Wahrscheinlichkeitsrechnung.

Der mittlere Codierungsaufwand je ursprünglichem Zeichen ist beim Code 1 gleich 2 Bits. Beim Code 2 ergibt sich ein mittlerer Aufwand von 0.5*1 + 0.25*2 + 0.125*3 + 0.125*3 = 1.75. Das ist deutlich weniger.

Man mache sich klar, welche wirtschaftlichen Konsequenzen eine derartige *Datenkompression* hat: Durch Datenkompression ist es beispielsweise möglich, die Dauer für die Übetragung einer DIN-A4-Seite beim Fernkopieren (Telefax) von etwa einer Viertelstunde unter eine Minute zu senken.

Ein Problem entsteht dadurch, dass die effizienteren Codes uneinheitliche Wortlänge haben. Was passiert, wenn man diese Wörter einfach hintereinanderschreibt, sozusagen ohne Punkt und Komma? Bei Blockcodes sind die Wortgrenzen aufgrund der konstanten Wortlänge immer erkennbar. Die Decodierung ist insofern einfach.

Ein Code heißt (eindeutig) *decodierbar*, wenn jeder Folge von Codezeichen, die durch Hintereinanderschreiben ohne Trennzeichen entsteht, immer eindeutig eine ursprüngliche Zeichenfolge zugeordnet werden kann.

Hätte man im Code 2 für das Zeichen b anstelle 10 das Codezeichen 01 gewählt, wären nach wie vor alle Codezeichen unterschiedlich. Allerdings wäre die eindeutige Decodierbarkeit von Codezeichenfolgen nicht mehr gegeben. Die Codezeichenfolge 01110 könnte dann sowohl zur ursprünglichen Zeichenfolge ada als auch zur ursprünglichen Zeichenfolge bc gehören.

Der Code 2 ist so konstruiert, dass dies Mehrdeutigkeiten nicht auftreten können. Der Code besitzt die sogenannte *Präfix-Eigenschaft:* Kein Codewort ist Anfang (Präfix) eines anderen. Diese Eigenschaft zieht die eindeutige Decodierbarkeit unmittelbar nach sich.

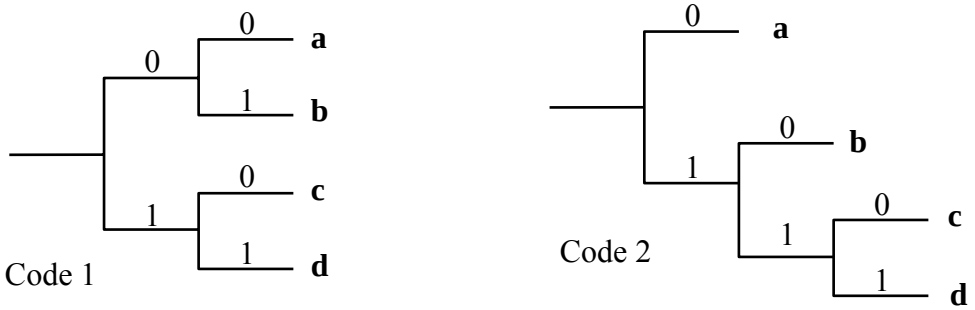


Bild 2.5-1 Codierbäume

Am *Codierbaum* machen wir uns die Präfix-Eigenschaft klar. **Bild 2.5-1** zeigt die Codierbäume der beiden Codes. Diese Codierbäume sind so zu interpretieren: Ausgehend von der Wurzel folgt man den Zweigen bis hin zu einem Blatt. Die Folge der an den Zweigen stehenden Bits ergibt das Codezeichen desjenigen Zeichens, das man schließlich erreicht. Ein Code mit Präfix-Eigenschaft zeichnet sich dadurch aus, dass die ursprünglichen Zeichen nur an den Blättern des Baumes stehen, und nicht an Verzweigungen.

Der Huffman-Code

Huffman-Codes sind Codes mit geringstmöglicher mittlerer Codewortlänge. Die Konstruktion eines solchen Codes läuft darauf hinaus, den Codierbaum - ausgehend von den Blättern - schrittweise aufzubauen.

Man wählt zunächst die beiden Zeichen mit den niedrigsten Wahrscheinlichkeiten aus. Die beiden Codewörter werden nur in der letzten Stelle unterschieden. Die beiden seltensten Zeichen sind damit zu den beiden Endpunkten einer Verzweigung des Codierbaums geworden. Der Verzweigungpunkt selbst wird nun als eigenes Zeichen aufgefasst. Die Wahrscheinlichkeit dieses Zeichens ergibt sich als Summe der Wahrscheinlichkeiten der beiden von ihm ausgehenden Zweige.

Nun ist ein Codierbaum für einen Code mit einer um eins verringerten Anzahl von Zeichen zu konstruieren. Der Schritt zur Reduktion der Zeichenanzahl lässt sich wiederholt anwenden. Man tut es solange, bis die Anzahl der Zeichen auf eins geschrumpft ist. Dann ist die Wurzel des Baums erreicht, und wir sind fertig. Nach der Codierung von Huffman erhält man beispielsweise den schon als günstig erkannten Code 2.

Die Entropie

Wir haben gesehen: Blockcodes sind nicht optimal. Unter dem Gesichtspunkt der Datenkompression sind Huffman-Codes günstiger, und die sind keine Blockcodes. Sind Huffman-Codes unter dem Aspekt der Datenkompression die bestmöglichen Codes?

Tatsächlich besitzen die Huffman-Codes eine sehr interessante Eigenschaft: Sie sind optimal in dem Sinne, dass unter allen eindeutig decodierbaren Codes keiner eine kürzere mittlere Codewortlänge hat.

Wir wollen diese Eigenschaft nicht beweisen. Aber der Grenzwert für die kürzestmögliche Codewortlänge soll noch allgemein angegeben werden. Er ist gegeben durch den mittlerem Informationsgehalt der Zeichen einer Quelle. Dieser ist folgendermaßen definiert.

Sei W ein Zeichenvorrat mit den Elementen w_1, w_2, ..., w_n. Diese Zeichen mögen mit den Wahrscheinlichkeiten p_1, p_2, ..., p_n auftreten - und zwar statistisch unabhängig voneinander. Der Informationsgehalt h der einzelnen Zeichen wird mit Hilfe des Zweierlogarithmus definiert: $h_k = -\text{ ld } p_k$. Der mittlere Informationsgehalt H je Zeichen, die sogenannte *Entropie*, ergibt sich durch die Mittelwertbildung über alle Zeichen:

$$H = p_1 \cdot h_1 + p_2 \cdot h_2 + ... + p_n \cdot h_n = -\Sigma_{1 \le k \le n}\, p_k \cdot \text{ld } p_k$$

Die Einheit des Informationsgehalts ist ein bit (von binary digit, Binärzeichen).

Dass die Entropie H tatsächlich ein geeignetes Maß für den Informationsgehalt ist, kann man sich an ein paar Beispielen und Sonderfällen leicht klar machen.

1. Wenn man die Zeichen des zweielementigen Zeichenvorrats B als gleichwahrscheinlich voraussetzt, dann ist der mittlere Informationsgehalt der Zeichen gleich $H = -1/2 \text{ ld}(1/2) - 1/2 \text{ ld}(1/2) = \text{ld } 2 = 1$ bit. Also: Ein Bit hat den Informationsgehalt von 1 bit.
2. Sei W ein Zeichenvorrat aus n Zeichen, die alle gleich wahrscheinlich sind. Dann gilt: $H = \text{ld } n$.
3. Sei W ein Zeichenvorrat gemäß Punkt 2. Die Entropie des Zeichenvorrats W^N bezeichnen wir mit H_N. Dann gilt: $H_N = N \cdot H_1 = N \cdot H$.
4. Die Formel aus Punkt 3 gilt auch dann noch, wenn die Zeichen nicht gleichwahrscheinlich sind.

Die Entropie ist eine untere Schranke für den Codieraufwand, die mittlere Codewortlänge m, eines Codes: $H \le m$.

Übung

2.5. 1 Konstruieren Sie den Codierbaum des Huffman-Codes zu folgenden Zeichen: a, b, c, d, e. Sie mögen mit den Wahrscheinlichkeiten 0.3, 0.24, 0.2, 0.15 und 0.11 auftreten.

2.5.2 Zeigen Sie für die bisher betrachteten Huffman-Codes, dass deren mittlere Codewortlängen m die Beziehung $H \le m < H+1$ erfüllen.

Die redundanzarme Codierung (Datenkompression) und insbesondere den Huffman-Code habe ich im BI-Bändchen „Codierungsverfahren“ beschrieben. Wer an weiteren Hintergründen interessiert ist, dem sei das Buch von Mildenberger empfohlen: Mildenberger, O.: Informationstheorie und Codierung. Vieweg, Braunschweig 1990.

2.6 Bäume

Programmtechnische Realisierung von Bäumen. Anwendung: Huffman-Code.

In diesem Abschnitt wird ein Programm vorgestellt, das den Codierbaum eines Huffman-Codes aufbaut. Die einzelnen Zeichen werden sukzessive zu Teilbäumen zusammengefasst. Wir beginnen also zunächst mit einem Baum für jedes ursprüngliche Zeichen. Diese Bäume werden über ihre Wurzeln identifiziert und zu einer zyklischen Liste verkettet. Der Zugang zu dieser Liste wird über den Cursor hergestellt: Er zeigt immer auf einen der Bäume. Er ist der „Zeiger im Wald".

Anfangs besteht jeder Baum nur aus einem einzigen Blatt (**Bild 2.6-1**, Teilbild I). Dann werden die beiden seltensten Zeichen zu einem neuen Baum zusammengefasst (Teilbild II). Das wiederholt sich dann (Teilbild III) bis schließlich der Codierbaum fertig ist (Teilbild IV).

Die Verkettung der Bäume untereinander geschieht mit demselben struct-Member next wie die Verkettung der Knoten eines Baumes. Man beachte, dass dieser Zeiger der Zweigrichtung entgegengesetzt ist. Also: next zeigt immer in Richtung Baumwurzel. Diese Darstellungsart für Bäume heißt *Parent-Repräsentation* (Elter-Darstellung).

Das Problem der Listenverwaltung habe ich hier mit einem „verzögerten Zeiger" gelöst: Der Cursor zeigt nicht auf das eigentlich interessierende Feld, sondern auf das davor. Durch diese indirekte Referenzierung wird der zweite Zeiger gespart. Allerdings ist nun die Referenzierung selbst etwas aufwendiger.

Übung: Warum wird das Ausdrucken der Codezeichen mit einer rekursiven Prozedur erledigt (WriteCode)? Geht es auch anders? Wenn ja, wie? Was sind die Vorteile und was die Nachteile der Verfahren?

Literaturhinweise

Das vorliegende Skriptum hat mehrere computergeeignete Darstellungsarten für Bäume gebracht. Besonders herausgehoben wurden die Postorder-Darstellung und die Parent-Repräsentation. Es gibt noch eine Fülle weiterer Möglichkeiten. Da Bäume spezielle Netze sind, kann man jede für Netze geeignete Darstellung auch für Bäume heranziehen. Das alles und eine Fülle von Algorithmen für Bäume - sowie Hinweise auf Anwendungen dieser Datenstruktur - sind in den Büchern von Aho/Hopcroft/ Ullman (1983) und Knuth (1973) zu finden.

Auch das Buch Klaus Neumann und Martin Morlock (Operations Research. Hanser, München 1993) bietet viele Algorithmen für Probleme, die sich auf netz- und baumartige Strukturen zurückführen lassen.

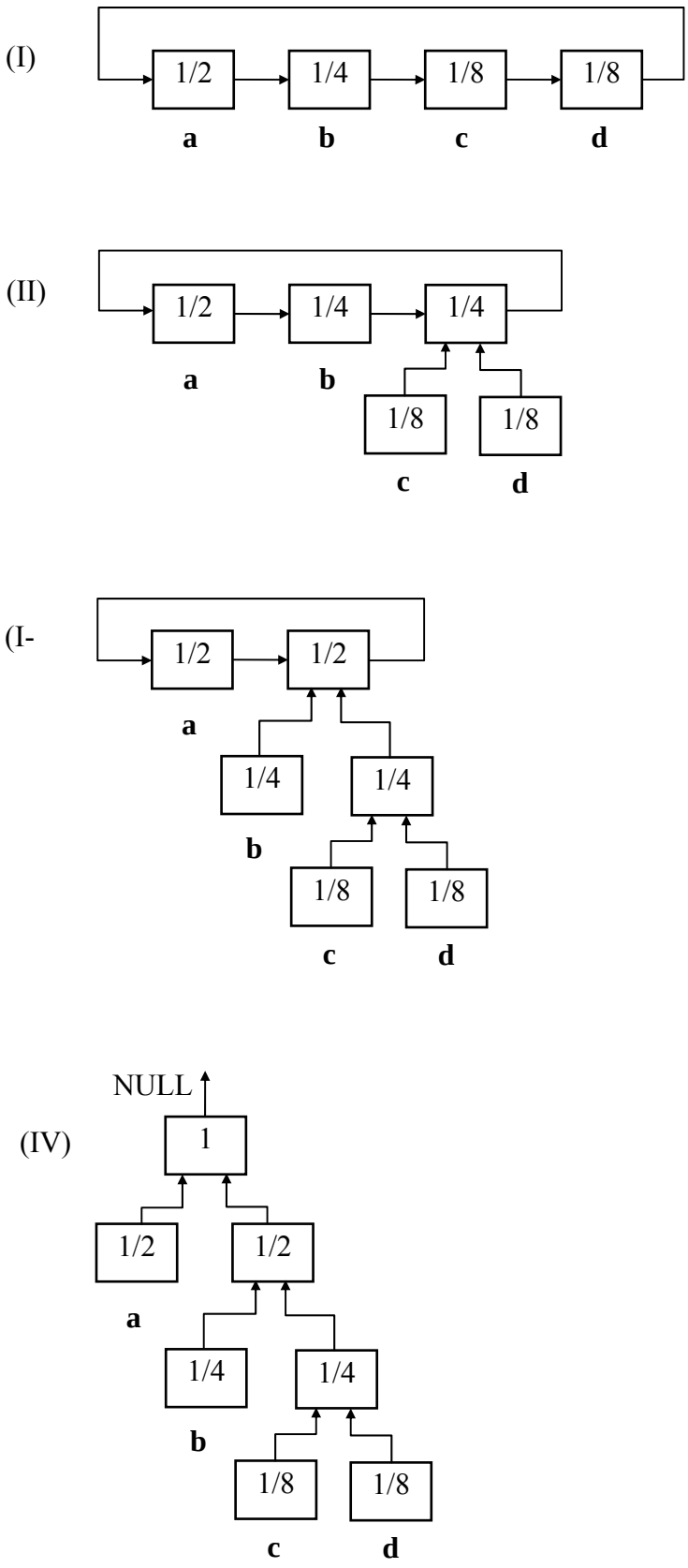


Bild 2.6-1 Konstruktion des Codierbaums

```
/*Huffman.c, Timm Grams, Fulda, 20.06.00 (15.06.01, 2.02.04)*/

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

#define Node struct Node
Node {
  float p;                  /*Gewicht, Wahrscheinlichkeit*/
  int sign;                 /*Elementarzeichen, Bit*/
  Node *next;               /*Verkettung von Baeumen und Knoten*/
};


Node *leaf[UCHAR_MAX+1];    /*Zuordnung Zeichen, Blatt*/
Node *cursor;               /*"Zeiger im Wald"*/
int n=0;                    /*Anzahl der Baeume*/

/*minimum() entfernt den Baum mit dem kleinsten Gewicht p aus dem Wald und
liefert als Rueckgabewert den Pointer auf ihn.
*************************************************************************/
Node* minimum(void){
  int i;
  Node *cur=cursor;
  for(i=1;i<n;i++){
    cur= cur->next;
    if(cur->next->p<cursor->next->p) cursor=cur;
  }
  cur=cursor->next;
  if (n>1) cursor->next=cursor->next->next; else cursor= NULL;
  n--;
  return cur;
}

/*insert(p) fuegt einen neuen Baum mit dem Gewicht p und dem Zeichen 0 in
den Wald ein und gibt den Pointer auf diesen Baum zurück.
*************************************************************************/
Node *insert(float p0){
  Node *h=(Node*)malloc(sizeof(Node));
  h->p=p0; h->sign=0;
  if(cursor==NULL) cursor=h->next=h; else {
    h->next=cursor->next;
    cursor->next=h;
  }
  n++;
  return h;
}

/*constructTree() baut den Codierbaum des Huffman-Codes auf.
*********************************************************/
void constructTree(void){
  Node *w1, *w2;
  while(n>1){
    w1=minimum(); w2=minimum();
    cursor=insert(w1->p+w2->p);
    w1->next= cursor;
    w2->next= cursor;
    w2->sign= 1;
  }
  cursor->next=NULL;
}


void writeCode(Node *x){            /*rekursive Funktion*/
  if(x->next!=NULL) {
    writeCode(x->next);
```

```
    printf("%1d", x->sign);
  }
}

void main(){
  float w, sw=0;                    /*Gewicht, Summe der Gewichte*/
  int i;                            /*Laufvariable*/
  int m=0;                          /*Anzahl der Zeichen*/
  unsigned char buf[80], c=0;

  /*Titel*/
  printf("HUFFMAN-CODE, ...);
  printf("Eingabe der darstellbaren Zeichen mit ihren Gewichten\n");

  /*Explizite Initialisierung unnoetig, da leaf extern deklariert ist*/

  /*Eingabe*/
  printf("? Anzahl der Zeichen = ");
  gets(buf); sscanf(buf, "%d", &m);
  for(i=1; i<=m; i++) {
    printf("? Zeichen Gewicht: ");
    gets(buf); sscanf(buf, "%c %e", &c, &w);
    leaf[c]=insert(w);
    sw+=w;
  }

  /*Verarbeitung*/
  if(m) constructTree();

  /*Ausgabe*/
  printf("! Code:\n");
  c=0;
  do if(leaf[c]) {
    printf("  %c(%.3f) -> ", c, leaf[c]->p/=sw);
    writeCode(leaf[c]);
    printf("\n");
  } while(c++<UCHAR_MAX);
}
```

Übung

2.6.1 Gehen Sie das Programm Huffman.c Schritt für Schritt durch und machen Sie sich die Bedeutung der einzelnen Anweisungen klar. Beantworten Sie dann die folgenden Fragen:

- Warum wird writeCode(…) Als rekursive Funktion definiert?
- Was ist der Grund für die häufige Verwendung von next beispielsweise im Ausdruck „cur->next->p<cursor->next->p“?
- Warum ist die Ausgabe als do-while-Schleife und nicht als while-Schleife organisiert?

2.7 Compilerbau*

Ein Parser für Kurzformausdrücke. Äquivalenz von Assembler- und While-Programmen. Funktionseinheiten eines Compilers. Phasen der Programmübersetzung. Historische Anmerkungen.

Ermittlung der Postorder-Darstellung direkt aus dem Ausdruck

Für die Ermittlung der Postorder-Darstellung eines Kurzform-Ausdrucks ist grundsätzlich nur ein Durchlauf der Zeichenfolge nötig. Je Verarbeitungsschritt wird dabei immer nur ein Zeichen des Kurzform-Ausdrucks betrachtet. Der hier darzustellende Algorithmus nutzt einen Stack und eine Zustandsvariable, die den gerade behandelten Knotentyp repräsentiert (Teil I, Abschnitt 2.4).

Der Syntaxbaum wird nicht in voller Größe aufgebaut. Die Einträge auf dem Stack stellen nur den für die weitere Entwicklung relevanten Ausschnitt des Syntaxbaums dar. Alles was für die weitere Entwicklung der Postorder-Darstellung nicht mehr relevant ist, wird wieder vom Stack entfernt, also vergessen.

Ausgehend vom Startsymbol wird für jedes Nichtterminal-Symbol, das nicht sofort abschließend behandelt werden kann, ein Eintrag in den Stack gemacht. Darin werden der Grad - das ist die Zahl der Operanden bzw. Kinder - und die Funktion des zugehörigen Knotens gespeichert. Anfangs wird die Zahl der Operanden (Kinder des Knotens) auf null und die Funktion gegebenenfalls auf undefiniert gesetzt.

Wird ein Verknüpfungszeichen angetroffen, so wird die entsprechende Funktion in den obersten Stackeintrag nötigenfalls nachgetragen.

Die Informationen für einen Knoten sind vollständig, wenn alle Kinder des Knotens bereits abgearbeitet worden sind. Ist das der Fall, dann werden die Informationen über den Knoten (Anzahl der Kinder und die Funktion) aus dem Stack genommen und in die Tabelle der Postorder-Darstellung eingetragen. Daraufhin werden die Informationen über den Elterknoten im Stack aktualisiert: Die Grad des Elterknotens wird um eins erhöht.

Die Postorder-Darstellung ist nach Abarbeitung des letzten Zeichens vollkommen aufgebaut.

Die Module Stack und Parser als Flowchart-Programme

Die am Ende dieses Abschnitts abgedruckten Programme zeigen, wie die Satzzerlegung komplizierter Sprachstrukturen auf einer einfachen Maschine mittels Assembler-Programmierung realisiert werden kann. Der Parser ist als Modul realisiert. Er benutzt einen ebenfalls als Modul realisierten Kellerspeicher (Stack).

Für die terminalen Symbole der Kurzformausdrücke werden die Ersatzzeichen gemäß **Tabelle 2.7-1** zugelassen. Da die Zeichen ≤ und ¬ den Zugriff auf einen speziellen Zeichensatz erfordern, werden auch in den Programmen grundsätzlich die entsprechenden Ersatzzeichen verwendet. Kleinbuchstaben in der Eingabe werden programmintern in Großbuchstaben umgewandelt.

Tabelle 2.7-1 Bezeichnung der Operatoren und Ersatzzeichen

Operator	Zeichen	Ersatzzeichen
IMP	≤	<
EQUAL	=	
OR	+	
AND	NULL	
NOT	¬	-

Die Module Stack und Parser sind zwar nicht in einer Assemblersprache geschrieben, aber bei genauem Hinsehen entpuppen sich die realisierten Funktionsstrukturen als Flowchart-Programme (Goto-Programme)[1]. Die C-Funktionen bilden also Assemblerprogramme nach. Die Übertragung in echte Assemblerprogramme ist leicht durch direkte Übersetzung möglich.

Da es hier darum ging, die Nähe zu den Assemblerprogrammen zu wahren und zu zeigen, dass man mit nur wenigen Sprachelementen auskommen kann, sind die Programmtexte keine besonders schönen C-Programme! Eine Satzzerlegung mittels C würde ausgiebig von Funktionen Gebrauch machen und die Technik der Rekursion nutzen. Jedem nichtterminalen Symbol wird dabei eine Funktion zugeordnet. Der Stack wird in diesem Fall dann implizit über die Funktionsaufrufe realisiert.

Compiler

Das Parser-Modul wandelt einen Ausdruck von einer Sprache (hier die Sprache der Kurzform-Ausdrücke) in eine andere (nämlich die Postorder-Darstellung). Auf der Ebene der Programmiersprachen nennt man ein solches Programm einen *Compiler* bzw. *Übersetzer*. Der Kurzformausdruck entspräche dann dem *Quellprogramm* und die Postorder-Darstellung dem *Zielprogramm* der Übersetzung.

Bei Compilern geht es meist darum, den Programmtext, der in einer höheren Programmiersprache - z.B. Pascal oder C - formuliert worden ist, in ein *Assemblerprogramm* bzw. ein *Maschinenprogramm* zu wandeln.

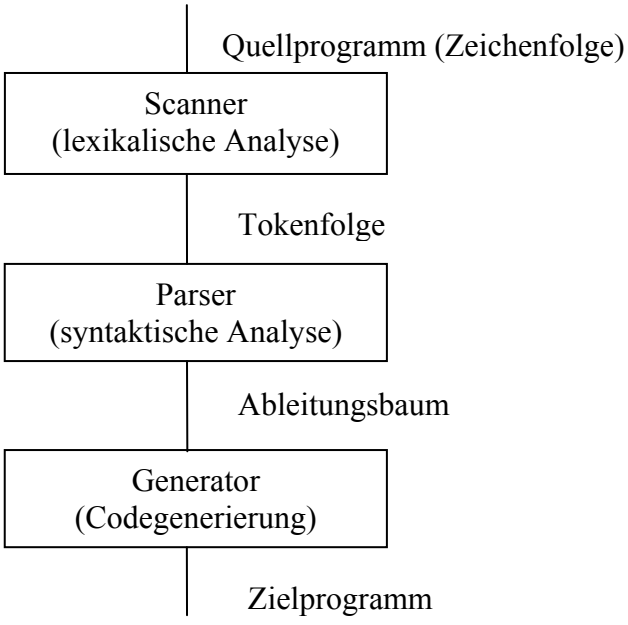


Bild 2.7-1 Phasen der Programmübersetzung und Funktionsmodule

[1] Flowchart-Programme kennen als einziges Strukturierungsmittel den (bedingten) Sprung, der in C mit der If-Anweisung und der Goto-Anweisung realisiert wird. While-Programme verzichten auf solche freien Sprünge und führen stattdessen die Strukturierungselemente Sequenz, Auswahl (if) und Schleife (while) ein. Dadurch, dass diese Elementen nur jeweils einen Zutrittspunkt und einen Austrittspunkt für den Programmablauf bieten, sollen schwer durchschaubare Spaghetti-Programme vermieden werden.

Auch bei solchen Compilern ist der *Parser* die zentrale Komponente. Seine Aufgabe ist die *syntaktische Analyse* des Quellprogramms und der Aufbau des *Syntaxbaums*. Genau das leistet auch unser Parser für die Kurzformausdrücke. Er liefert die Postorder-Darstellung der Syntaxbäume. Statt Syntaxbaum heißt es im Compilerbau auch *Ableitungsbaum*

Das Quellprogramm ist eine Folge von Zeichen aus der Codetabelle. Diese Zeichen sind in der Kurzform-Darstellung gleichzeitig die lexikalischen Einheiten, also die „Bedeutungsträger" der Sprache. In C setzen sich die lexikalischen Einheiten der Sprache teilweise aus mehreren Zeichen zusammen: *Schlüsselwörter* wie while und do, *Zahlen*, *Zeichenketten, Bezeichner* und *Spezialsymbole* wie +=, /* und <=.

Solche Compilern sind demnach um ein Modul für die *lexikalische Analyse* zu ergänzen. Dieses Modul wird *Scanner* genannt. Es erfasst die lexikalischen Einheiten und gibt sie an den Parser weiter. Die lexikalischen Einheiten heißen *Token*.

Nicht immer ist die Darstellung des Syntaxbaums das Ziel der Übersetzung. Normalerweise ist der Syntaxbaum erst in die Zielsprache zu übertragen. Meistens ist das der Code, den der Prozessor versteht. Das Modul für die *Codegenerierung* wird auch kurz *Generator* genannt.

Das Blockdiagramm eines Compilers ist in **Bild 2.7-1** dargestellt.

Interpreter (Prozessor)

In unserem Fall ist das Zielprogramm - die Postorder-Darstellung des Kurzform-Ausdrucks - kein Programm, das auf dem Computer direkt lauffähig ist. Für die Ausführung dieses Programms brauchen wir einen eigenen *Interpreter*. Der Interpreter übernimmt also die Rolle des Prozessors. Der im Anschluss wiedergegebene Interpreter für Kurzformausdrücke in Postorder-Darstellung ist eine Realisierung des Auswertungsalgorithmus, der im Abschnitt 2.4 skizziert worden ist.

Literaturhinweise und historische Anmerkungen

Broy, M.; Denert, E. (Eds.): Software Pioneers. Contributions to Software Engineering. Springer-Verlag, Berlin, Heidelberg 2002

Unter dem Stichwort *Übersetzer* stellt der Informatik-Duden die wesentlichen Funktionen eines Compilers zusammen. Das Buch von Niklaus Wirth (1986) ist eine knappe und doch umfassende Einführung in den Compilerbau von einem der einflussreichsten Experten auf diesem Gebiet. (Niklaus Wirth ist Schöpfer der Programmiersprachen Pascal, Modula 2 und Oberon.)

Die zentrale Rolle, die der Stack für die Architektur von Programmiersprachen und für den Compilerbau spielt, wurde von Friedrich L. Bauer herausgestellt. Er hat dafür sogar ein Patent erhalten. Dieses wohl erste Patent für Software wurde durch eine Hardware-Realisierung des Stacks in einem der frühen Computer möglich. Geschichtliche Anmerkungen von Friedrich L. Bauer selbst und Abdrucke der Originaldokumente sind in den „Software Pioneers" von Broy und Denert enthalten.

Dort findet man auch den berühmten Aufsatz von Edsgar W. Dijkstra „Go To Statement Considered Harmful", der die Vorteile der Strukturierten Programmierung auf der Grundlage von While-Programmen gegenüber der Flowchart-Programmierung hervorhebt. Übrigens bekennt Niklaus Wirth in eben diesem Buch, dass die Beibehaltung der Goto-Anweisung in der von ihm geschaffenen Programmiersprache Pascal ein Fehler war.

Dass While-Programme und der Verzicht auf die Goto-Anweisung die programmtechnischen Möglichkeiten gegenüber den Flowchart-Programmen nicht einschränken, ist die Aussage des

sogenannten *Strukturtheorems* (Linger, Mills, Witt, S. 118). Der Beweis dieses Satzes beruht auf derselben Technik, die in den Programmtexten Stack.c und Parser.c (am Schluss dieses Kapitels) angewendet wird.

Übung

2.7.1 Gehen Sie den Programmablauf des Parsers für den Kurzform-Ausdruck A+B+C≤¬A¬B¬C einmal mit dem Papiercomputer durch.

Stack

```
/*stack.h, Timm Grams, Fulda, 11.01.03
  Das Modul stack realisiert ein Stack von Einzelzeichen (char).
  Die Rueckgabewerte von push() und pop() sind ungleich null, wenn die
  Operation durchgefuehrt worden ist.
*************************************************************/
#include <stdio.h>

extern void resetStack();
extern int push(char);
extern int pop(char*);

/*remv(i) entfernt die i obersten Elemente vom Stack (remove)
***********************/
extern void remv(int);

/*get(i) liefert den den Zeiger auf diejenige Variablen, die i Plaetze
unterhalb von TOP steht. Ergebnis des Funktionsaufs *get(0) ist TOP.
Falls der Stack leer ist, wird der Nullpointer uebergeben.
*********************/
extern char *get(int);

/*printsStack(stdout) liefert Ausdruck des Stacks auf der Konsole
************************/
extern void printStack(FILE*);

/*stack.c, Timm Grams, Fulda, 11.01.03*/

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

#define MAX 1000
char stack[MAX];
int SP=-1;

void resetStack() {SP=-1;}

int push(char x) {
  if (++SP<MAX) {stack[SP]=x; return 1;}
```

```
  else return 0;
}


int pop(char *y) {
  if(SP<0) return 0;
  else {*y=stack[SP--]; return 1;}
}

void remv(int i) {SP-=i; if (SP<-1) SP=-1;}

char *get(int i) {
  if (SP-i<0) return 0;
  else return &stack[SP-i];
}

void printStack(FILE *out) {
  int i;
  fprintf(out, "|");
  for (i=0; i<=SP; i++) fprintf(out,"%c", stack[i]);
  fprintf(out, ">");
}
```

Parser

```
/*Parser.h, Timm Grams, Fulda, 11.01.03

  Das Modul Parser bietet eine Parser an, der fuer einen Kurzformausdruck
  die Postorder-Dastellung des Syntax-Baumes liefert. Im Aufruf
  parser(expr, tree) wird auf expr der Kurzformausdruck erwartet und tree
  ist das Array der Postorder-Darstellung. Letztere hat den Aufbau
  (Grad, Funktion, Grad, Funktion, ..., Grad, Funktion). Da Nullen im Array
  vorkommen, kann dieses Array NICHT als C-String aufgefasst werden!

  Falls parse(expr, tree) erfolgreich abgeschlossen wird, ist der
  Ergebniswert gleich der Laenge der Postorder-Darstellung, das ist der
  Index hinter dem letzten Funktionseintrag. Bei nicht erfolgreichem
  Abschluss ist der Ergebniswert  gleich null. In diesem Fall wird auf dem
  String errorMessage die Bezeichnung des Knotens uebermittelt, bei dem der
  Fehler aufgetreten ist.  Die Position des Kurzformausdrucks, auf der das
```

```
  fehlerhafte Zeichen steht, ist gleich errorPos.
**************************************************************************/
extern int parse(const char[], char[]);

extern char *errorMessage;
extern int errorPos;

/*Parser.c, Timm Grams, Fulda, 11.01.03*/

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

#define MAX 1000
#define error(pos, message) errorPos=pos; errorMessage=message; return 0
/*Eine sehr heikle Sache. Ersatzloesung fuer die Funktionsschachtelung
  bei Pascal. Der Funktionsaufruf sollte, abgesehen von gut begruendeten
  Ausnahmen, in geschweiften Klammern stehen.*/

char *t, *a, c;
char *errorMessage;
int errorPos;

enum {function, degree};
typedef enum {false, true} bool;
typedef enum {Expression, SimpleExpression,
              Term, Factor, Element, Negation} Node;
int n, i; /*Number of Tree Symbols, input index*/

void next() {c=*++a; ++i;}
int isElement(char c) {return c=='0'||c=='1'||'A'<=c&&c<='Z';}
int isOp(char c) {return c=='='||c=='<';}
int isFactor(char c) {return c=='-'||c=='('||isElement(c);}

void submit() {
  if(1<*get(degree)||'-'==*get(function)) {
    *t++=*get(degree); *t++=*get(function);
    n+=2;
  }
  remv(2);
}
```

```
/*parse merkt sich nur die gerade relevanten Teile des Syntaxbaums. Der Baum
wird quasi gleichzeitig erstellt, durchlaufen und in den nicht mehr
interessierenden Teilen gleich wieder "vergessen". Zur Speicherung der gerade
relevanten Teile genuegt der Stack.
****************************************/
int parse(const char expr[], char tree[]) {
  Node node=Expression; /*Laufvariable des Postorder-Durchlaufs*/
  bool establish=true;  /*bedeutet, dass der Knoten erstellt werden muss*/
  n=i=0; t=tree; resetStack();

  c=*(a=expr);
  while (true) {
   //printf("\nnode: %d establish: %d i: %d n: %d c: %c", node, establish, i, n, c);
   switch (node) {

    case Expression:
      if(establish) {push(0); push('?'); node=SimpleExpression;}
      else if(isOp(c) && *get(function)=='?') {
        *get(function)= c; node=SimpleExpression; establish=true; next();
      } else if (*get(degree)) {
        submit();
        if (c==0)
          if (get(0)) {error(i, "Expression");} else return n;
        else if(c==')' && get(degree)) {
          ++*get(degree); node=Factor; next();
        } else {error(i, "Expression");}
      } else {error(i, "Expression");}
      break;

    case SimpleExpression:
      if(establish) {push(0); push('+'); node=Term;}
      else if(c=='+') {node=Term; establish=true; next();}
      else if (*get(degree)) {
        submit(); ++*get(degree); node=Expression;
      } else {error(i, "SimpleExpression");}
      break;

    case Term:
      if(establish) {push(0); push('&'); node=Factor;}
      else if(isFactor(c)) {node=Factor; establish=true;}
      else if (*get(degree)) {
        submit(); ++*get(degree); node=SimpleExpression;
```

```
      } else {error(i, "Term");}
      break;

    case Factor:
      if (establish)
        if(c=='-') node=Negation;
        else if(c=='(') {node=Expression; next();}
        else if(isElement(c)) {
          *t++=0; *t++=c; n+=2; next();
          ++*get(degree);
          establish=false;
        } else establish=false;
      else if (*get(function)=='-') node=Negation;
      else node=Term;
      break;

    case Negation:
      if(establish) {push(0); push('-'); next(); node=Factor;}
      else if (*get(function)=='-')
        if(*get(degree)) {submit(); ++*get(degree);}
        else {error(i, "Negation");}
      else {node=Factor; establish=true;}
      break;

    default: error(i, "Default");
   }
   //printf(" stack: ", c); printStack(stdout);
  }
}
```

Interpreter

```
/*Intrprtr.h, Timm Grams, Fulda, 14.01.03

  eval(tree, n) liefert als Rueckgabewert den Wert des variablenfreien
  Kurzformausdrucks tree, der in Postorderdarstellung der Laenge n vorliegt.
  Die Eingabe wird als korrekt vorausgesetzt - Fehlerkontrollen des
  Kurzformausdrucks werden nicht durchgefuehrt.
*********************************************************************/
extern int eval(const char[], char);
```

```
/*Intrprtr.c, Timm Grams, Fulda, 14.01.03*/

#include "stack.h"

int eval(const char t[], int n) {
  int i, j;
  char c, acc;

  resetStack();
  for (i=0; i<n; i+=2) {
    if(t[i]==0) push(t[i+1]);
    else switch (t[i+1]) {
      case '=': if(pop(&acc)&&pop(&c)) push((c==acc)+'0'); else return -1;
                break;
      case '<': if (pop(&acc)&&pop(&c)) push((c<=acc)+'0'); else return -2;
                break;
      case '+': acc=0;
                for(j=0; j<t[i]; j++)
                  if(pop(&c)) acc|=c-'0'; else return -3;
                push(acc+'0');
                break;
      case '&': acc=1;
                for(j=0; j<t[i]; j++)
                  if(pop(&c)) acc&=c-'0'; else return -4;
                push(acc+'0');
                break;
      case '-': if(pop(&c)) push('1'+'0'-c); else return -5;
                break;
    }
  }
  pop(&c);
  if (get(0)) return -6; else return c-'0';
}
```

LogScope

```
/*LogScope.c, Timm Grams, Fulda, 11.01.03
  Das Programm LogScope berechnet den logischen Spielraum von Kurzform-
  Ausdruecken, die in der Textdatei in.txt angegeben sind. Ausgabedatei
  ist out.txt. Dort stehen die Kurzformausdruecke zusammen mit ihrer
```

```
  Postorderdarstellung und ihrem logischen Spielraum.
**********************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include "Parser.h"
#include "Intrprtr.h"
#define MAX 1000

void prnt(FILE *out, char s[], int n) {
  int i;
  for (i=0; i<n; i++) fprintf(out, "%c", s[i]);
}

int inc(char v[], int n) {
  char i=n, c=1;
  while (i--) {
    v[i]+=c;
    if ('1'<v[i]) {c=1; v[i]='0';} else c=0;
  }
  i=n; while (i--) if (v[i]=='1') return 1;
  return 0;
}


void main() {
  char s[80], c, v[27];
  int i, j, n;
  long z, zmax, m;
  FILE *in, *out;
  char tree[MAX], t[MAX];
  int nTree;

  if ((in=fopen("in.txt", "r"))==NULL) {
    printf("Fehler beim Oeffnen zum Lesen"); exit(0);
  }
  if ((out=fopen("out.txt", "w"))==NULL) {
    printf("Fehler beim Oeffnen zum Schreiben"); exit(0);
  }

  while(fscanf(in, "%s", s)!=EOF) {
    char *p, *q;
    for (p=s; *p; p++) *p=toupper(*p);
```

```
    fprintf(out, "\n Kurzform: %s \nPostorder: ", s);

    if (nTree=parse(s, tree))
      for (i=0; i<nTree; i++)
        if (i%2) fprintf(out, "%c", tree[i]);
      else fprintf(out, "%d", tree[i]);
    else {
      for (i=0; i<errorPos; i++) fprintf(out, " ");
      fprintf(out, "^ %s", errorMessage);
    }

    /*s: Sequence of Variables
      v: Sequence of values
    ************************/
    n=0;
    for (c='A'; c<='Z'; c++)
      for (j=1; j<nTree; j+=2)
        if(tree[j]==c) {s[n]=c; v[n++]='0'; break;}

    for (i=0; i<nTree; i++) t[i]=tree[i];

    fprintf(out, "\n LogScope: "); prnt(out, s, n);
    do {
      for (i=0; i<n; i++)
        for (j=1; j<nTree; j+=2)
          if (tree[j]==s[i]) t[j]=v[i];
      if (eval(t, nTree)) {
        fprintf(out, "\n           ");
        //for (i=0; i<nTree; i++) fprintf(out, "%c", t[i]);
        prnt(out, v, n);
      }
    } while(inc(v,n));
  }
  fclose(in); fclose(out);
}
```

Anhang: Standard-Bibliothek (ANSI Runtime Library)

Die folgenden Funktions-Prototypen, Typen und symbolischen Namen (Makros) sind in der *ANSI Runtime Library* der Programmiersprache C definiert (Darnell/Margolis, 1991, 405-488, Groß, 1998).

Alle Funktionsnamen der Bibliothek sind reservierte Bezeichner und dürfen nicht neu deklariert oder definiert werden. Gleichfalls sind alle mit einem Unterstrich beginnenden Bezeichner für solche Makros reserviert, die „hinter den Kulissen" verwendet werden.

Zusätzlich zu den eigentlichen Funktionen enthält die Runtime Library noch eine Reihe von Header Dateien. Jede Funktion verlangt eine oder mehrere dieser Dateien. Diese Dateien müssen in der Quelltextdatei immer dann erscheinen, wenn diese Funktion aufgerufen wird.

Ein-/Ausgabe (<stdio.h>)

Standard Ein-/Ausgabe-Funktionen

```
int  fclose (FILE *);
int  fflush (FILE *);
FILE *fopen (const char *, const char *);
FILE *freopen (const char *, const char *, FILE *);
int  setbuf (FILE *, char *);
int  setvbuf (FILE *, char *, int, size_t);
```

Funktionen für die formatierte Ein-/Ausgabe

```
int  fprintf (FILE *, const char *, ...);
int  fscanf (FILE *, const char *, ...);
int  printf (const char *, ...);
int  scanf (const char *, ...);
int  sprintf (char *, const char *, ...);
int  sscanf (char *, const char *, ...);
int  vfprintf (FILE *, const char *, va_list);
int  vprintf (const char *, va_list);
int  vsprintf (char *, const char *, va_list);
```

Funktionen für die Ein-/Ausgabe von Einzelzeichen

```
int  fgetc (FILE *);
char *fgets (char *, int, FILE *);
int  fputc (int, FILE *);
int  fputs (const char *, FILE *);
int  getc (FILE *);
int  getchar (void);
char *gets (char *);
int  putc (int, FILE *);
int  putchar (int);
int  puts (const char *);
int  ungetc (int, FILE *);
```

Funktionen für den Direktzugriff

```
int  fseek (FILE *, long, int);
long ftell (FILE *);
void rewind (FILE *);
```

Funktionen zur Fehlerbehandlung

```
void clearerr (FILE *);
```

```
int  feof (FILE *);
int  ferror (FILE *);
void perror (const char *);
```

Die Formatangaben der *printf*-Funktionsfamilie

Bei *printf* und *vprintf* wird als Ausgabekanal *stdout* benutzt (i.a. der Bildschirm). Die Ausgabe eines nicht-vollen Puffers kann erzwungen werden durch `fflush(stdout);`. Das Format (der erste Parameter bei *printf* und *vprintf* bzw. der zweite Parameter bei *fprintf*, *sprintf*, *vfprintf* und *vsprintf*) gibt an, wie viele Argumente folgen (durch die Anzahl der Formatangaben) und wie sie formatiert werden sollen.

```
Format = "[ Text ] [Formatangabe] ..."
Formatangabe = % [-] [+] [space] [#] [width] [.precision] [prefix]
               type
```

Die Parameter dürfen Ausdrücke sein, z.B.

```
printf ("%d * %d = %d\n", 2, 2, 2 * 2);
```

Die Bedeutung der Symbole ist in der folgenden Tabelle 1 zusammengefasst; *width* und *precision* können als „*“ spezifiziert sein. In diesem Fall wird ihr Wert berechnet und muß als nächstes Argument (Typ: int) in der Parameterliste stehen. Beispiel:

```
printf ("%%*d\t%*d\n", 5, 23);  (Feldbreite: 5, Wert: 23)
```

Als Ergebnis liefert *printf* die Anzahl der ausgegebenen Zeichen zurück bzw. einen negativen Wert, falls ein Fehler erkannt wurde.

Die Formatangaben der *scanf*-Funktionsfamilie

Bei *scanf* und *vscanf* wird als Eingabekanal *stdin* benutzt (i.a. die Tastatur). Das Format (Position analog zur *printf*-Familie) gibt an, wie viele Argumente eingelesen und wie sie formatiert werden sollen. Als Parameter müssen *die Adressen von Variablen* übergeben werden, in die die Werte gespeichert werden sollen.

```
Format = "[ Text ] [Formatangabe] ..."
Formatangabe = % [*] [width] [prefix] type
```

Die Bedeutung der Symbole ist in der Tabelle 2 zusammengefasst. Das Format kann Texte und Formatangaben enthalten. Es wird erwartet, dass *normale* Texte (von Trennzeichen verschiedene Zeichen) mit Texten der Eingabe übereinstimmen. Ein Trennzeichen innerhalb des Formats bewirkt, dass alle Trennzeichen der Eingabe überlesen werden. Beispiel:

```
scanf (" Wert: %d", &n);
```

Führende Trennzeichen (Leerzeichen, Tabulatoren, ...) werden überlesen. Dann wird die Zeichenfolge "Wert:" erwartet, danach eventuelle Trennzeichen und anschließend eine Dezimalzahl, deren Wert in der Variablen *n* gespeichert wird. Falls nach den ersten Trennzeichen nicht die Zeichenfolge "Wert:" folgt, meldet *scanf* einen Fehler und bricht den Lesevorgang ab.

Tabelle 1 Bedeutung der Felder in der Formatangabe der *printf*-Funktionsfamilie

%	Anfang einer Formatangabe (%% gibt ein Prozentzeichen aus)
-	linksbündige Ausgabe
+	alle Zahlen werden mit "+" oder "−" Zeichen ausgegeben
space	alle negativen Zahlen mit "−" Zeichen und alle positiven Zahlen mit " " ausgeben. *Default:* negative Zahlen: "−", positive Zahlen: weder "+" noch " "
#	abhängig vom Typ c, d, i, s, u: keine Bedeutung o: als erste Ziffer "0" ausgeben x, X: der Zahl "0x" bzw "0X" voranstellen e, E, f, g, G: Dezimalpunkt ausgeben (z.B. auch: 3.) g, G: anhängende Nullen ausgeben (z.B. 3.00)

width	minimale Feldbreite. Bei kleineren Zahlen werden Füllzeichen ausgegeben (i.a. Leerzeichen). Falls die Feldbreite mit einer Null beginnt, wird das Feld ggf. mit "0" aufgefüllt. Falls die Feldbreite zu klein gewählt wird, um die Zahl aufzunehmen, wird sie automatisch vergrößert (eine Zahl wird niemals auf die Feldbreite reduziert). Beispiel:
	`printf ("%%4d\t%4d\n", 25);`
	`printf ("%%04d\t%04d\n", 25);`
	`printf ("%%04d\t%04d\n", 25000);`

.precision	Für *Gleitkommazahlen* gibt das Feld *precision* die Anzahl der Ziffern nach dem Dezimalpunkt an. Für *ganze Zahlen* gibt es die minimale Anzahl Zeichen an, die ausgegeben werden sollen. Für *Zeichenfolgen* gibt es die maximale Anzahl Zeichen an, die ausgegeben werden dürfen.
prefix	h short int, unsigned short int (beim Typ: d, i, o, x, X, u) l long int, unsigned long int (beim Typ: d, i, o, x, X, u) L long double (beim Typ: e, E, f, g, G)
type	d, i Dezimalzahl mit Vorzeichen u Dezimalzahl ohne Vorzeichen o Oktalzahl ohne Vorzeichen x, X Hexadezimalzahl ohne Vorzeichen (x: abcdef; X: ABCDEF) c einzelnes Zeichen (char) s Zeichenfolge (char *) f Gleitkommazahl (Festkommazahl) in Dezimaldarstellung: [-]mmm.ddd (*Default*: 6 Zeichen nach dem Dezimalpunkt. Bei einer Genauigkeit von ".0" wird der Dezimalpunkt nicht ausgegeben.) e, E Gleitkommazahl in Exponentialdarstellung: [-]m.ddd{**e, E**}{+,-}xx (*Default*: 6 Zeichen Genauigkeit. Bei einer Genauigkeit von ".0" wird der Dezimalpunkt nicht ausgegeben.) g, G Wenn der Exponent kleiner als -4 oder größer gleich der Genauigkeit ist, wird die Zahl im %e bzw. %E Format ausgegeben und sonst im %f Format. Der Dezimalpunkt und / oder anhängende Nullen werden nach Möglichkeit nicht ausgegeben. p implementierungs-abhängige Ausgabe eines Zeigers (pointer; void *) % "%" ausgeben

Tabelle 2 Bedeutung der Felder in der Formatangabe der *scanf*-Funktionsfamilie

%	Anfang einer Formatangabe (%% liest ein Prozentzeichen ein)	
*	Zuweisung unterdrücken. ..*scanf* liest zwar den angegebenen Wert, weist ihn aber keiner Variablen zu, d.h. die Parameterliste, darf für diese Formatangabe keine Variablenadresse enthalten.	
width	maximale Feldbreite (im Gegensatz zur minimalen Feldbreite der *printf*-Familie). ..*scanf* liest höchstens *width* Zeichen ein, konvertiert sie und weist sie ggf. einer Variablen zu.	
prefix	h	short int, unsigned short int (beim Typ: d, i, n, o, x, X, u)
	l	long int, unsigned long int (beim Typ: d, i, n, o, x, X, u)
	l	double (beim Typ: e, E, f, g, G)
	L	long double (beim Typ: e, E, f, g, G)
type	d	liest eine Dezimalzahl ein (Parameter: `int *`)
	i	liest eine Dezimalzahl ein, die ein Prefix und / oder Suffix enthalten darf. Erlaubte Prefixe: -, +, 0x, 0X, 0 (Oktalzahl). Erlaubte Suffixe: u, U (unsigned int) l, L (long int). Parameter: Adresse einer Variablen des entsprechenden Typs.
	u	liest Dezimalzahl ohne Vorzeichen ein (Parameter: `unsigned int *`)
	o	liest Oktalzahl ohne Vorzeichen ein (Parameter: `unsigned int *`)
	x, X	liest Sedezimalzahl ohne Vorzeichen mit oder ohne führendem 0x bzw. 0X ein. Parameter: `unsigned int *`
	c	liest abhängig von der Feldbreite (*Default*: 1) ein oder mehrere Zeichen ein. Trennzeichen (siehe Kapitel 2.1) sind in diesem Fall normale Zeichen. Hinter den gelesenen Zeichen wird keine "\0" angefügt! Das nächste von einem Trennzeichen verschiedene Zeichen kann durch "%1s" eingelesenwerden. Falls mehrere Zeichen eingelesen werden, muß das Feld groß genug sein, um alle Zeichen aufzunehmen (es werden keine Grenzen überprüft **!**). (Parameter: `char *`)
	s	liest eine von Trennzeichen verschiedene Zeichenfolge ein. Hinter den gelesenen Zeichen wird eine "\0" angefügt. Das Feld muß groß genug sein um alle Zeichen aufzunehmen (es werden keine Grenzen überprüft **!**). Parameter: `char *.`
	e, E, f, g, G	liest eine Gleitkommazahl ein, die aus folgenden Komponenten bestehen darf: einem optionalen Vorzeichen, einer Folge von Ziffern mit höchstens einem Dezimalpunkt, einem optionalen Exponenten, der mit *e* bzw. *E* eingeleitet wird und eine Dezimalzahl mit Vorzeichen enthalten kann. Parameter: `float *`
	p	liest einen Zeiger in der Form von *printf* (*"%p"*) ein. Parameter: `void *`
	[...]	liest eine Zeichenfolge ein, die nur Zeichen aus der angegebenen Liste enthält. Hinter den gelesenen Zeichen wird eine "\0" angefügt. Das Feld muß groß genug sein, um alle Zeichen aufzunehmen (es werden keine Grenzen überprüft **!**). Falls das Zeichen "`]`" direkt auf die öffnende Klammer folgt, gehört es mit zur Liste (`[]...]`). Falls das Zeichen "-" in der Liste vorkommt und nicht das erste oder letzte Zeichen der Liste ist, ist dessen Interpretation implementierungs-abhängig (i.a. bezeichnet es einen Bereich von Zeichen, z.B.: a-z entspricht a bis z). Parameter: `char *`
	[^...]	liest eine Zeichenfolge ein, die nur Zeichen enthält, die nicht zur angegebenen Liste gehören. Hinter den gelesenen Zeichen wird eine "\0" angefügt. Das Feld muß groß genug sein, um alle Zeichen aufzunehmen (es werden keine Grenzen überprüft **!**). Falls das Zeichen "`]`" direkt auf das Zeichen "^" folgt, gehört es mit zur Liste (`[^]...]`). Falls das Zeichen "-" in der Liste vorkommt und nicht das zweite oder letzte Zeichen der Liste ist, ist dessen Interpretation implementierungs-abhängig (i.a. bezeichnet es einen Bereich von Zeichen, z.B.: a-z entspricht a bis z). Parameter: `char *`
	%	liest ein Prozentzeichen. Es erfolgt keine Zuweisung.

Standard-Funktionen (<stdlib.h>)

```
typedef struct { int quot; int rem; } div_t;
typedef struct { long quot; long rem; } ldiv_t;
```

String Conversion

```
double atof (const char *);
int atoi (const char *);
long atol (const char *);
double strtod (const char *, char **);
long strtol (const char *, char **, int);
unsigned long strtoul (const char *, char **, int);  /* fehlt bei GNU C
*/
```

(Pseudo-)Zufallszahlengeneratoren

```
int rand (void);
void srand (unsigned int);
```

Speicherverwaltung

```
void *calloc (size_t, size_t);
void free (void *);
void *malloc (size_t);
void *realloc (void *, size_t);
```

Kommunikation mit der Umgebung

```
void abort (void);
int atexit (void (*)(void));
void exit (int);
char *getenv (const char *);
int system (const char *);
```

Suchen und Sortieren

```
void *bsearch (const void *, const void *, size_t, size_t,
               int (*)(const void *, const void *));
void qsort (void *, size_t, size_t,
            int (*)(const void *, const void *));
```

Integer-Arithmetik

```
int abs (int);
```

Zeichenklassifikation (<ctype.h>)

```
int isalnum (int); /* check if a character is a numeral or letter  */
int isalpha (int); /* check if a character is a letter             */
int iscntrl (int); /* check if a character is a control character  */
int isdigit (int); /* check if a character is a numeral            */
int isgraph (int); /* check if a char. is printable (except space) */
int islower (int); /* check if a character is a lower-case letter  */
int isprint (int); /* check if a char. is printable (incl. space)  */
int ispunct (int); /* check if a character is a punctuation mark   */
int isspace (int); /* check if a character is white space          */
int isupper (int); /* check if a character is a upper-case letter  */
int isxdigit (int); /* check if a char. is a hexadecimal numeral    */
int tolower (int); /* convert character to lower case              */
```

```
int toupper (int); /* convert character to upper case              */
```

Manipulation von Zeichenfolgen (<string.h>)

Kopierfunktionen

```
void *memcpy (void *, const void *, size_t);
void *memmove (void *, const void *, size_t);     /* fehlt bei GNU C */
char *strcpy (char *, const char *);
char *strncpy (char *, const char *, size_t);
```

Verkettungsfunktionen

```
char *strcat (char *, const char *);
char *strncat (char *, const char *, size_t);
```

Vergleichen

```
int memcmp (const void *, const void *, size_t);
int strcmp (const char *, const char *);
int strncmp (const char *, const char *, size_t);
```

Suche

```
void *memchr (const void *, int, size_t);
char *strchr (const char *, int);
size_t strcspn (const char *, const char *);
char *strpbrk (const char *, const char *);
char *strrchr (const char *, int);
size_t strspn (const char *, const char *);
char *strstr (const char *, const char *);
char *strtok (char *, const char *);
```

Verschiedene Funktionen

```
void *memset (void *, int, size_t);
size_t  strlen (const char *);
```

Datum und Zeit (<time.h>)

```
typedef unsigned long size_t;
typedef long clock_t;
typedef unsigned long   time_t;

struct tm { int tm_sec;        /* seconds after the minute [0-59]      */
            int tm_min;        /* minutes after the hour [0-59]        */
            int tm_hour;       /* hours since midnight [0-23]          */
            int tm_mday;       /* day of the month [1-31]              */
            int tm_mon;        /* months since January [0-11]          */
            int tm_year;       /* years since 1900                     */
            int tm_wday;       /* days since Sunday [0-6]              */
            int tm_yday;       /* days since January 1 [0-365]         */
            int tm_isdst;      /* Daylight Savings Time flag           */
          };

double difftime (time_t, time_t);
time_t mktime (struct tm *);
time_t time (time_t *);
char *asctime (const struct tm *);
char *ctime (const time_t *);
struct tm *gmtime (const time_t *);
struct tm *localtime (const time_t *);
```

Beispiel 1:

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

int main (void)
{
  clock_t zeit;

  zeit = clock ();
  printf ("Zeitmessung laeuft. Ende: <return>\n");
  (void) getchar ();
  zeit = clock () - zeit;
  printf ("Zeitdauer: %lu \"Ticks\" bzw. %.2f Sekunden.\n",
     zeit, (float) zeit / CLOCKS_PER_SEC);
  return 0;
}
```

Beispiel 2:

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

int main (void)
{
  time_t t1, t2;

  time (&t1);
  printf ("Zeitmessung laeuft. Ende: <return>\n");
  (void) getchar ();
  time (&t2);
  printf ("Zeitdauer: %.2f Sekunden.\n", difftime (t2, t1));
  return 0;
}
```

Mathematische Funktionen (<math.h>)

Winkelfunktionen

```
double  acos (double);
double  asin (double);
double  atan (double);
double  atan2 (double, double);
double  cos (double);
double  sin (double);
double  tan (double);
```

Hyperbelfunktionen

```
double  cosh (double);
double  sinh (double);
double  tanh (double);
```

Exponentialfunktion und Logarithmus

```
double  exp (double);
double  frexp (double, int *);
double  ldexp (double, int);
double  log (double);
double  log10 (double);
double  modf (double, double *);
```

Weitere mathematische Funktionen

```
double  pow (double, double);
double  sqrt (double);
double  ceil (double);
double  fabs (double);
double  floor (double);
double  fmod (double);
```

Minimal- und Maximalwerte ganzer Zahlen (<limits.h>)

```
#define CHAR_BIT        8
#define CHAR_MIN        (-128)
#define SCHAR_MIN       (-128)
#define CHAR_MAX        127
#define SCHAR_MAX       127
#define UCHAR_MAX       255
#define SHRT_MIN        (-32768)
#define SHRT_MAX        32767
#define USHRT_MAX       65535
#define INT_MIN         (-2147483647-1)
#define INT_MAX         2147483647
#define UINT_MAX        4294967295U
#define LONG_MIN        (-2147483647L-1L)
#define LONG_MAX        2147483647L
#define ULONG_MAX       4294967295UL
```

Minimal- und Maximalwerte von Gleitkommazahlen (<float.h>)

```
#define FLT_ROUNDS      1
#define FLT_RADIX       2
#define FLT_MANT_DIG    24
#define FLT_DIG         6
#define FLT_EPSILON     1.19209290e-07F
#define FLT_MIN_EXP     (-125)
#define FLT_MIN_10_EXP  (-37)
#define FLT_MAX_EXP     128
#define FLT_MAX_10_EXP  38
#define FLT_MIN         1.17549435e-38F
#define FLT_MAX         3.40282347e+38F

#define DBL_MANT_DIG    53
#define DBL_DIG         15
#define DBL_EPSILON     2.2204460492503131e-16
#define DBL_MIN_EXP     (-1021)
#define DBL_MIN_10_EXP  (-307)
#define DBL_MAX_EXP     1024
#define DBL_MAX_10_EXP  308
#define DBL_MIN         2.2250738585072010e-308
#define DBL_MAX         1.7976931348623167e+308

#define LDBL_MANT_DIG   53
#define LDBL_DIG        15
#define LDBL_EPSILON    2.2204460492503131e-16L
#define LDBL_MIN_EXP    (-1021)
#define LDBL_MIN_10_EXP (-307)
#define LDBL_MAX_EXP    1024
#define LDBL_MAX_10_EXP 308
#define LDBL_MIN        2.2250738585072010e-308L
#define LDBL_MAX        1.7976931348623167e+308L
```

Sachverzeichnis

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z