Kurzeinführung in die Programmiersprache C

für Programmierer von Script- und anderen Hochsprachen
auf der Basis des Standards ISO/IEC 9899:1990

Bernd Rosenlecher

(1.3.8) Generiert am 11. Dezember 2015. Dieses Dokument wurde mit TEX/LATEX gesetzt.

Inhaltsverzeichnis

Danksagung: Ich danke ...

... u.a. den in der Literatur genannten Autoren für die Erweiterung meines Wissens,

Cord Eggers, der mich in die Prägnanz und Eleganz von C einführte,

Wolf Lammen (LUG-Balista) für das Lesen einer Vorfassung und Ermunterung,

Helmut Schönborn (HMH) für Ermunterung und Förderung auch dieser Arbeit,

Hayo Schmidt (HMH), der die Mühe des sorgfältigen Korrekturlesens der fast endgültigen Fassung auf sich nahm, so manche Fehler fand, die mir entgangen waren, und wertvolle Hinweise sachlicher und sprachlicher Art geben konnte.

Alle noch verbleibenden Fehler und Unschönheiten fallen selbstverständlich in meine Verantwortung.

br

1 Vorwort

Dieser Text entstand als Bezugsrahmen zu einer kompakten Einführung in die Programmiersprache C, die auf Wunsch einiger Mitglieder der Hamburger Linux User Group LUG–Balista, sowie der Hamburger Microcomputer Hochschulgruppe e.V. (HMH), von Januar bis März des Jahres 2007 als wöchentlicher Arbeitskreis im Brakula (Bramfelder Kulturladen) in Hamburg stattfand.

Auf Grund der sehr knapp bemessenen Zeit konnte natürlich nur ein grober Abriß der Sprache dargeboten werden. Dieser Text kann und soll also die Lektüre z.B. des K&R2, m.E. der immer noch besten Einführung in die Sprache C, nicht ersetzen, sondern eher als Motivation zu weiteren eigenen Studien dienen. Hierzu ist vor allem die Auswahl an grundlegender und weiterführender Literatur in der Bibliographie gedacht. Die Anführung der englischen Fachausdrücke soll den Zugang dazu erleichtern.

Als Ausgangspunkt wurde der z.Z. am weitesten verbreitete C-Dialekt gewählt, der sich am Standard ISO/IEC 9899:1990 unter Einbeziehung des »Amendment 1« von 1994 orientiert. Auf die Behandlung vieler Einzelheiten (u.a. wide und multibyte characters, noch gültige alte Syntax, formatierte Eingabe mit scanf(), so manche syntaktische Feinheiten, modulare Programmierung etc.) wurde bewußt verzichtet. Stattdessen wurden eine Kurzeinführung in das C-Compiliersystem unter Unix, sowie eine ausführlichere Darstellung der Repräsentationen von Daten und Code eines Programms auf Maschinenebene für notwendig erachtet, um die Visualisierung des C-Systems – besonders für Script- und Hochsprachenprogrammierer, die ja meist weit entfernt von der zugrundeliegenden Hardware arbeiten – zu erleichtern. Ebenso soll die Beifügung der Quelltexte einiger kleiner C-Programme einen Einblick in die Struktur von C ermöglichen und Beispiele für die klassische Formatierung bieten.

Als Programmier- und Entwicklungsumgebung für diese Einführung wurde GNU/Linux gewählt, jedes andere Unix-artige System (BSD, Solaris, Mac OS X, AIX, Irix ... etc. pp.) mit jedem beliebigen dort vorhandenen C-System wäre aber genauso geeignet. Speziell die GNU Entwicklungsumgebung ist wegen ihrer Qualität auch auf andere Systeme (DOS: DJGPP, Windows: Cygwin) portiert worden, und kann dort ebenso eingesetzt werden. Auf die Besonderheiten dieser Systeme wird aber nicht weiter eingegangen. Ganz bewußt wurde die klassische Entwicklungsmethode auf Shell-Ebene gewählt, also keine IDE, da dies neben der Unabhängigkeit von solchen Produkten auch einen besseren Einblick in die gesamte Programmentwicklung gewährleistet.

Warum würde jemand heutzutage die Sprache C erlernen wollen, die trotz relativer Kleinheit und einfachen Aufbaus aufgrund mancher historisch gewachsener Inkonsistenzen und einer gewöhnungsbedürftigen Deklarationssyntax so manche Tücken und Gefahren birgt? Wahrscheinlich wegen ihrer Omnipräsenz, um in C dargestellte Algorithmen zu verstehen, um selbst einmal dieses mächtige Werkzeug der System- und Anwendungsprogrammierung nutzen zu können, oder einfach, um aus dem riesigen Schatz in C geschriebener, frei im Quelltext zugänglicher Software Anregungen für eigene Projekte zu beziehen. Viel Spaß und Erfolg dabei!

Bernd Rosenlecher
Hamburg, im Juli 2009

2 Einleitung

Die Sprache C gehört zur ALGOL-Familie der sog. imperativen Programmiersprachen. Dennis M. Ritchie entwarf C ungefähr 1972, als er eine hocheffiziente, flexible und maschinennahe Sprache brauchte, um das zwei Jahre vorher in den Bell Labs entstandene Betriebssystem UNIX auf die DEC PDP11 zu portieren. Nach Veröffentlichung des Buches »The C Programming Language« von Kernighan und Ritchie (daher der Name K&R[1] C) und der weiteren Verbreitung des Unix-Systems nicht nur an Universitäten trat C seinen Siegeszug als System- und Anwendungssprache vom Kleinst- bis zum Supercomputer ab Mitte der 80er Jahre rund um die Welt an. Die zeitliche Entwicklung von C incl. ihrer Vorgänger läßt sich so darstellen:

1960	ALGOL 60
1963	CPL (Cambridge)
1967	BCPL (M. Richards)
1970	B (K. Thompson)
1972	C (D.M. Ritchie), interne Verwendung bei AT&T
1978	K&R C (Kernighan/Ritchie)
1989	ANSI C Standard (X3J11, ANSI X3.159-1989), K&R2
1990	ISO C Standard (JTC1 SC22 WG14, ISO/IEC 9899:1990)
1994	Technical Corrigendum 1 (TC1) und Amendment 1
1995	Technical Corrigendum 2 (TC2)
1998	finale Entwurfsfassung des ANSI/ISO Komitees
1999	neuer Standard ISO/IEC 9899:1999

Die Sprache C zeichnet sich durch folgende Eigenschaften aus:

- relativ *kleiner* Sprachkern, kompakte Notation
- reichhaltiger Satz von Standarddatentypen
- reichhaltiger Satz von Operatoren
- Zeiger, Felder, Verbünde für komplexe Datenstrukturen
- gute Abbildung dieser auf Maschinenebene: hohe Effizienz
- alles andere – E/A, Speicherverwaltung etc. in Std.-Bibliothek
- wegen Einfachheit und Verbreitung extrem hohe Portabilität

Die immense Flexibilität und Ausdrucksstärke von C birgt aber auch große Gefahren in der Hand eines unerfahrenen oder leichtfertigen Programmierers: C ist *keine sichere* Sprache! Größte Disziplin ist geboten, um Fehler zu vermeiden. Der Leitsatz von Ritchie beim Entwurf der Sprache lautete: »Trust the programmer!«

[1] Die zweite Ausgabe – K&R2 – ist im Anhang unter [Kern89] angeführt.

3 Das C-Entwicklungssystem

Das C-Entwicklungssystem unter Unix besteht traditionellerweise aus mehreren Programmen (Prinzip: Werkzeugkasten), die in modularer Form die Bearbeitung von gemischten Programm-Quelltexten bis zur Erzeugung von Programmbibliotheken oder der lauffähigen Version eines Programmes übernehmen. Die wichtigsten Komponenten – mit ihren traditionellen Namen – sind:

`cc`	C-Compiler-Kontrollprogramm	(C compiler driver)
`cpp`	C-Präprozessor	(C preprocessor)
`cc1`	eigentlicher C-Übersetzer	(C compiler, 1st stage)
`as`	Assembler	(assembler)
`ld`	Binder	(linker, orig. loader)

Ferner gibt es noch eine Reihe weiterer Dienstprogramme, die helfen, Quelltexte und Bibliotheken zu verwalten, sowie größere, modulare Programmierprojekte zu organisieren. Ohne im weiteren darauf einzugehen, seien einige davon mit ihren Zwecken hier aufgeführt:

`gdb`	GNU Debugger	(gnu debug)
`nm`	Symboltabelle ausgeben	(name map)
`size`	Sektionsgrößen listen	(...)
`strings`	Zeichenketten extrahieren	(...)
`strip`	Objektsymboltabelle entfernen	(...)
`objdump`	Disassembler und mehr	(...)
`readelf`	ELF Format anzeigen	(...)
`file`	Art einer Datei anzeigen (raten)	(...)
`gprof`	Profilmessungen ermöglichen	(gnu profiler)
`ar`	(statisches) Archiv verwalten	(archiver)
`ranlib`	Index für Archiv erzeugen	(randomize lib)
`ld.so`	dynamischer Binder	(loader for shared objects)
`cb`	Quelltext formatieren, einrücken	(c beautifier)
`indent`	Quelltext formatieren, einrücken	(...)
`flex`	GNUs Lexer	(fast lex)
`yacc`	Parsergenerator	(yet another compiler compiler)
`bison`	GNUs Ersatz für yacc	(Wortspiel)
`make`	Übersetzung, Bindung organisieren	(...)

Der GNU C Compiler kann auch als gcc aufgerufen werden. Bei neueren Versionen des GNU Compilers ist `cpp` auch in `cc1` integriert, das braucht uns aber nicht weiter zu stören, da wir alle Komponenten über `cc` bedienen werden. `cc` ist nicht der C-Compiler, sondern das Steuerprogramm des ganzen Übersetzungssystems, dem Optionen und die zu bearbeitenden Dateien als Parameter übergeben werden und das dann nach Bedarf die einzelnen Komponenten mit ihren zugehörigen Optionen aktiviert.

`cc` erkennt die Art der ihm zugeführten Dateien an ihren Endungen, die unter Unix konventionsgemäß so aussehen:

`xxx.c`	C-Quelltext
`xxx.i`	Zwischendatei, von `cpp` erzeugt
`xxx.s`	Assembler-Quelltext, von `cc1` erzeugt oder selbst geschrieben
`xxx.S`	Assembler-Quelltext, der noch von `cpp` bearbeitet werden muß
`xxx.o`	Object-Code, Binärdatei im Prozessor-Format, von `as` erzeugt
`a.out`	Default-Name für ausführbares Programm, von `ld` erzeugt

cc kennt eine Fülle von Optionen oder Schaltern, die den Verlauf der weiteren Verarbeitung steuern. Einige hier verwendete sind:

`cc --help`	einige häufig gebrauchte Optionen listen (gcc-spezifisch)
`cc -v ...`	verboser Modus (gcc-spezifisch)
`cc -c ...`	nur compilieren und Object-Code (`.o`) erzeugen
`cc -S ...`	nur compilieren und Assembler-Code (`.s`) erzeugen
`cc <file>.c`	ausführbares Programm namens `a.out` erzeugen
`cc -o <ofile> ...`	Ausgabedatei mit Namen `ofile` erzeugen

Weitere Optionen nebst einer Fülle relevanter Informationen zur Benutzung findet man online im Manual (man gcc), bzw. im GNU-Infosystem (info gcc) oder in Papierform (darin kann man sich auch Notizen machen) in [GNUCC].

4 Lexikalische Elemente

Der Grundzeichensatz für C-Quelltexte umfaßt folgende sichtbare Zeichen:

Großbuchstaben	`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`
Kleinbuchstaben	`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`
Dezimalziffern	`0 1 2 3 4 5 6 7 8 9`
Unterstrich	`_`
Interpunktion	`! " # % & ' ( ) * + , - . / : ; < = > ? [ \ ] ^ { | } ~`

Zusätzlich können folgende Zeichen vorkommen:

Zeichen	Bedeutung	Ersatzdarstellung
space	Leerzeichen	–
BEL	Alarmglocke (bell)	`\a`
BS	Rückschritt (backspace)	`\b`
FF	Seitenvorschub (form feed)	`\f`
NL	Zeilenvorschub (newline)	`\n`
CR	Wagenrücklauf (carriage return)	`\r`
HT	Horizontaltabulator (horizontal tab)	`\t`
VT	Vertikaltabulator (vertical tab)	`\v`

Es gibt auch Ersatzdarstellungen für die Anführungszeichen und zwei weitere Sonderzeichen zur Verwendung in Zeichen- und Zeichenkettenkonstanten (siehe Ab-

schnitt 4.5, Seite 9). Hier dient der Rückschrägstrich dazu, die Sonderbedeutung des betr. Zeichens zu unterdrücken: `\"`, `\'`, `\?`, `\\`. Um alle Zeichen des Zeichensatzes der Maschine darstellen zu können, gibt es ferner sog. numerische Escape-Sequenzen (Ersatzdarstellungen):

`\d`, oder `\dd` oder `\ddd`	d (1...3) ist Oktalziffer (oft gbraucht: `'\0'`, die Null)
`\xh` oder `\xhh` oder ...	h (beliebige Anzahl) ist Hexadezimalziffer

Für Entwicklungsumgebungen, die bestimmte Zeichen nicht darstellen können, gibt es noch sog. Tri- und Digraphen, die kaum je gebraucht werden und mit denen wir uns daher auch nicht weiter aufhalten wollen.

C unterscheidet einen Quellzeichensatz (*source character set*) und einen Zielzeichensatz (*target character set*). Wenn die Übersetzungsumgebung mit der Ausführungsumgebung identisch ist, sind beide gleich. In Zeichen- und Zeichenkettenkonstanten (auch Literale genannt) können alle Zeichen des verwendeten Systems vorkommen. Außerdem können diese Zeichensätze von lokalen Einstellungen – der sog. *locale* abhängig sein. Wenn das Programm startet, arbeitet es mit dem Zeichensatz der sog. `"C"` *locale*. Zugehörige Makros sind definiert in `locale.h`.

4.1 Leerraum

Als Leerraum (*white space*) gelten Leerzeichen, Zeilenvorschub, Wagenrücklauf, vertikaler und horizontaler Tabulator, sowie Seitenvorschub. Kommentare gelten auch als Leerraum. Leerraum wird syntaktisch ignoriert, außer in Zeichenketten- oder Zeichenkonstanten; er dient dazu, sonst aneinandergrenzende Wörter, Zeichen etc. zu trennen und den Quelltext für Menschen durch übersichtliche Gestaltung, z.B. Einrückungen nach Kontrollstruktur etc., gut lesbar zu machen.

4.2 Kommentare

Kommentare werden durch die Zeichenpaare `/*` und `*/` erzeugt. Alles, was dazwischen steht – auf einer Zeile oder mit beliebig vielen Zeilen dazwischen, gilt als Kommentar. Kommentare dürfen nicht geschachtelt werden. `/*` *Das ist zum Beispiel ein Kommentar ... und hier geht er immer noch weiter* `*/`.

4.3 Schlüsselwörter

C hat die folgenden 32 Schlüsselwörter (*reserved words*):

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

4.4 Bezeichner

C-Bezeichner (*identifier*), sonst auch schlicht *Namen* genannt, werden folgendermaßen gebildet (als regulärer Ausdruck in Unix-Notation):

```
[A-Za-z_][A-Za-z_0-9]*
```

d.h. Buchstabe oder Unterstrich, optional gefolgt von beliebiger (auch Null) Folge eben dieser, inklusive der Ziffern.

Bezeichner dürfen nicht mit einer Ziffer beginnen, Groß- und Kleinbuchstaben sind als verschieden zu werten. Bezeichner dürfen nicht aus der Menge der o.g. Schlüsselwörter sein (oder aus der Menge von Namen, die für die Standardbibliothek reserviert sind, sie müssen sich mindestens in den ersten 31 Zeichen unterscheiden. Mit Unterstrich beginnende Namen sind für das System reserviert und sollten nicht verwendet werden. Bezeichner mit externer Bindung (d.h. Weiterverarbeitung durch Linker etc.) können weiteren Beschränkungen unterliegen.

4.5 Konstanten

C kennt vier Hauptgruppen von Konstanten:

Ganzzahlkonstanten	Dezimal-, Oktal- oder Hex-Darstellung
Gleitpunktzahlkonstanten	mit Dezimalpunkt und/oder Exponentkennung
Zeichenkonstanten	eingeschlossen in `'...'`
Zeichenkettenkonstanten	eingeschlossen in `"..."`

Numerische Konstanten sind immer positiv, ein etwa vorhandenes Vorzeichen gilt als unärer Operator auf der Konstanten und gehört nicht dazu. Ganzzahlkonstanten sind vom Typ `int`, wenn das nicht ausreicht, vom Typ `long`, wenn auch das nicht ausreicht, vom Typ `unsigned long`. Man kann die größeren Typen auch durch Anfügen von Suffixen erzwingen, wie aus der folgenden Tabelle ersichtlich. Beginnt die Ganzzahlkonstante mit `0x` oder `0X`, so liegt Hexnotation vor und es folgen eine oder mehrere Hexziffern. Dabei stehen A-F bzw. a-f für die Werte 10...15. Beginnt andernfalls die Ganzzahlkonstante mit einer `0`, so liegt Oktalnotation vor und es folgen eine oder mehrere Oktalziffern, andernfalls liegt Dezimalnotation vor.

Gleitpunktzahlkonstanten sind immer vom Typ `double`, falls nicht durch Suffix als `float` oder `long double` gekennzeichnet. Zur Erkennung müssen mindestens der Dezimalpunkt oder die Exponentkennung vorhanden sein.

Dezimalziffern	`0 1 2 3 4 5 6 7 8 9`
Oktalziffern	`0 1 2 3 4 5 6 7`
Hexziffern	`0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f`
`0`	Die Konstante 0 (Null)
`l L`	Ganzzahlsuffix für `long` (Verwechselungsgefahr l mit 1!)
`u U`	Ganzzahlsuffix für `unsigned`
`f F l L`	Gleitpunktzahlsuffix für `float` bzw. `long double` (s.o.)
`e E`	Gleitpunktzahlkennung für Exponent

Eine Zeichenkonstante (*character constant*) ist ein in einfache Hochkommata eingeschlossenes Zeichen aus dem Zeichensatz oder seine (auch mehrere Zeichen umfassende) Ersatzdarstellung. Die Betrachtung sog. *wide character constants*, sowie sog. *multi byte character constants* unterbleibt hier. Zeichenkonstanten sind vom Typ `int`, dürfen aber nicht wertmäßig größer als der entspr. Typ `char` sein. Das kommt bei vernünftiger Anwendung obigen Rezepts aber kaum vor.

Eine Zeichenkettenkonstante (*string constant*) ist eine in sog. doppelte Anführungszeichen eingeschlossene Zeichenkette auf einer Zeile. Sie darf alle Zeichen des Zeichensatzes, incl. etwaiger Ersatzdarstellungen, und (dann signifikanten) Leerraum enthalten. Nur durch Leerraum getrennte Zeichenketten werden vom Präprozessor zusammengefügt und gelten als eine Zeichenkette. Man kann eine Zeile auch umbrechen, indem man sie mit einem Rückschrägstrich terminiert. Die auf diese Weise fortgeführte Zeile gilt dann als *eine* logische Zeile.

Zeichenketten werden standardgemäß als *array of* `char` von niederen zu höheren Adressen mit terminierendem Nullwert im Speicher abgelegt. Ihre Speichergröße ist daher immer um 1 größer als die Größe, die der Anzahl der enthaltenen Zeichen entsprechen würde. Das sind also die allseits verbreiteten sog. *C-Strings*. Der Nullwert dient als Terminierungsmarke für alle Routinen der Standardbibliothek und kann folglich im String selbst nicht vorkommen. Der terminierende Nullwert gehört somit nicht zu den Zeichen des Strings und wird folglich bei Ermittlung seiner Länge auch nicht mitgezählt.

Eine Zeichenkette als Typ *array of* `char` zu sehen, nimmt man aber nur bei der Initialisierung von Arrays oder der Anwendung des `sizeof`-Operators wahr. Bei den meisten Verwendungen treten jedoch sofort die üblichen syntaktischen Umwandlungen von C in Kraft, und man sieht nur noch einen Zeiger auf das erste Zeichen, also den Typ `char*`, über den man dann alle weitere Verarbeitung steuern kann.

5 Der C-Präprozessor

Dem C-Präprozessor obliegt die Vorverarbeitung des C-Quelltexts zur sog. Übersetzungseinheit (*translation unit*), die dann dem eigentlichen Compiler zur Weiterverarbeitung übergeben wird. Er arbeitet als zeilenorientierte Textersetzungsmaschine und versteht die C-Syntax nicht. Seine Aufgabe ist es, jede Zeile mit einem *newline character* abzuschließen, unabhängig von der äußeren Form einer Textzeile, Trigraphen durch ihre Entsprechungen zu ersetzen, Zeilen, die mit einem Rückschrägstrich enden, zusammenzufügen, Zeichengruppen zu ersetzen (z.B. Escape-Sequenzen, Makros), Leerraum zu kondensieren, Kommentare (`/* ... */`) zu entfernen und durch ein Leerzeichen zu ersetzen, Direktiven auszuführen (auch wiederholt und rekursiv), und Dateien einzufügen (mit denen er dann rekursiv das gleiche anstellt).

Präprozessordirektiven werden mit `#` eingeleitet. Sie beginnen traditionell am linken Rand und stehen auf *einer* logischen Zeile. Es gibt folgende Direktiven:

`#include <datei.h>`	Standard-Header hier einfügen
`#include "datei.h"`	eigenen Header hier einfügen
`#define DIES jenes 17`	überall `DIES` durch `jenes 17` ersetzen, sog. Makro
`#undef XXX`	Makrodefinition `XXX` entfernen
`#line 47`	nächste Zeilennummer in der Datei
`#error "some failure!"`	zur Compilierzeit Fehlermeldung erzeugen
`#pragma builtin(xyz)`	implementationsdefinierte Option
`#ifdef FEATURE`	bedingte Compilierung
`#ifndef FEATURE`	bedingte Compilierung
`#if`	bedingte Compilierung
`#elif`	bedingte Compilierung
`#else`	bedingte Compilierung
`#endif`	bedingte Compilierung
`defined`	optional zur Verwendung mit `#if` und `#elif`

Präprozessor-Makronamen, wie oben z.B. `DIES`, `XXX` und `FEATURE`, werden traditionsgemäß meist komplett in Großbuchstaben geschrieben, führende Unterstriche sind für das System reserviert und sollten nicht verwendet werden.

Zur Zeichenkettenerzeugung, -umwandlung und -zusammenfügung im Zusammenhang mit der Makroverarbeitung gibt es schließlich noch die Operatoren `#` und `##`, mit denen man allerlei trickreiche Dinge anstellen kann und die hier nicht weiter behandelt werden. Aus gutem Grund! Früher wurde der Präprozessor sehr extensiv angewandt, dabei wurden die Quelltexte, d.h. was der Programmierer zu Gesicht bekam, oft in mehreren Stufen der rekursiven Textersetzung und Makroexpansion verändert. Wenn dann bei solch *raffinierten* Konstruktionen bei der Wartung, Weiterentwicklung, oder auch nur nach Aufspielen neuer Systemdateien plötzlich Syntaxfehler auftraten, waren diese oft auch mittels der besten Compiler-Fehlerdiagnostik nur äußerst schwer zu beheben, weil alle diese *Zauberei* ja stattfindet, bevor der eigentliche Compiler die so veränderte Übersetzungseinheit überhaupt zu fassen bekommt. Schlimmer wäre noch, wenn keine Syntaxfehler gemeldet werden, aber auf diese Weise erzeugte semantische Fehler zu »unerklärlichen« Laufzeitfehlern und »seltsamem« Programmverhalten führten.

Man vergesse nie: Der Präprozessor ist eine reine Textersetzungsmaschine, ohne jegliche Kenntnis von C! Semantische Klarheit von Quellcode hat heutzutage jedoch höchste Priorität. Daher ist der moderne Trend in der Anwendungsentwicklung (ca. seit Beginn der 90er Jahre), den Präprozessor nur noch für einfache Makros[2] sowie Inklusion und – falls notwendig – bedingte Compilierung einzusetzen. In der Systemsoftware sieht es allerdings etwas anders aus, wie man leicht beim Studium der Headerdateien (siehe im folgenden Abschnitt) feststellen kann. Man traut den Systemprogrammierern offenbar mehr Durchblick und Disziplin zu!

[2] wenn überhaupt: B. Stroustrup – der Schöpfer von C++ – brachte, an Cato angelehnt, seinen Lieblingssatz: »Übrigens bin ich der Meinung: Der C-Präprozessor muß verschwinden!« Soweit ist es denn doch nicht gekommen – er vollbringt ja immerhin auch noch andere nützliche Dinge.

6 Die Standardbibliothek

Während es bei früheren Programmiersprachen allgemein üblich war, die Bedienung der Peripherie, Ein- und Ausgabebefehle, Formatieranweisungen für den Druck, spezielle, für den prospektiven Anwendungsbereich erforderliche mathematische oder textverarbeitende Funktionen und ähnliches alles in der Sprache selbst unterzubringen, wurde C von Anfang an ausgelegt, einen möglichst kleinen Sprachkern in Verbindung mit einer Standardbibliothek zu verwenden. Die Sprache sollte es dem Benutzer auf einfache Weise ermöglichen, diese Bibliothek seinem Bedarf anzupassen und auf Wunsch beliebig zu erweitern. Diese Entwurfsphilosophie ist eines der Hauptkennzeichen von C geblieben.

Zur Sprache C gehört eine Standardbibliothek (*standard C library*), deren Programmierschnittstelle (*API*)[3] über die weiter unten aufgelisteten, insgesamt 18 sog. Header-Dateien[4] definiert wird. Die zugehörige Objektbibliothek befindet sich traditionsgemäß unter `/usr/lib`, wo sie der Linker finden kann. Diese Dateien sind meist unter dem Verzeichnis `/usr/include/...` abgelegt, und das C-System bekommt die betreffenden *Standardfundorte* für Header-Dateien und Objektbibliotheken bei der Installation eincompiliert. Die Header können (und sollten) natürlich gelesen werden: Sie enthalten die Definitionen für Makros und Datentypen, sowie die Deklarationen von Namen und Funktionen in den entspr. Abschnitten der Bibliothek.

`<assert.h>`	`<ctype.h>`	`<errno.h>`	`<float.h>`	`<limits.h>`	`<locale.h>`
`<math.h>`	`<setjmp.h>`	`<signal.h>`	`<stdarg.h>`	`<stddef.h>`	`<stdio.h>`
`<stdlib.h>`	`<string.h>`	`<time.h>`	`<iso646.h>`	`<wchar.h>`	`<wctype.h>`

Die Header `<iso646.h>`, `<wchar.h>` und `<wctype.h>` sind erst mit dem sog. *Amendment 1*, 1994, dazugekommen.

Die Anbindung entspr. Abschnitte der Standardbibliothek sollte im Quelltext immer über die Einbindung der jeweils zutreffenden Header mittels der `#include <...>` Präprozessoranweisungen geschehen, um die notwendigen Definitionen alle korrekt zu übernehmen.

Die Einbindung der entspr. Teile der Objektbibliothek durch den Linker geschieht meist automatisch, zuweilen ist es jedoch notwendig, bestimmte Teile – oft die zu `math.h` gehörenden Funktionen – mittels spezieller Linkeroptionen – in diesem Falle `-lm` – bei der Compilierung explizit anzufordern.

C unterscheidet die Bibliothek betreffend zwei Arten von Implementation: *hosted* und *free standing implementation*. Erstere muß den vollen Satz der 18 Header zur Verfügung stellen, letztere (z.B. gilt das für Systemkernentwicklung und für sog. *embedded systems*) braucht nur die Header `<float.h>`, `<limits.h>`, `<stdarg.h>` und `<stddef.h>` anzubieten, um dem Standard in dieser Beziehung zu genügen.

[3] Application Programming Interface

[4] Es müssen übrigens keine wirklichen *Dateien* sein, der Standard spricht von *entities*, d.h. der Implementation ist es freigestellt, die Wirkung auch auf andere Weise zu erreichen, z.B. indem im Compiler ein magischer Schalter umgelegt wird.

7 Syntaktische Elemente

7.1 Datentypen

Der Begriff des Datentyps beinhaltet folgendes:

- die Größe und Ausrichtung des belegten Speicherplatzes (*size, alignment*)
- die interne Darstellung (Bitbelegung)
- die Interpretation und damit den Wertebereich
- die darauf anwendbaren bzw. erlaubten Operationen

ISO-C verfügt über einen reichhaltigen Satz von Datentypen, die sich wie in folgender Übersicht gezeigt organisieren lassen:

C-Typbezeichnung	Gruppe	Klasse	Kategorie	MinBit
`char`	integer	arithmetic	scalar	8
`signed char`	integer	arithmetic	scalar	8
`unsigned char`	integer	arithmetic	scalar	8
`short`, `signed short`	integer	arithmetic	scalar	16
`unsigned short`	integer	arithmetic	scalar	16
`int`, `signed int`	integer	arithmetic	scalar	16
`unsigned int`	integer	arithmetic	scalar	16
`long`, `signed long`	integer	arithmetic	scalar	32
`unsigned long`	integer	arithmetic	scalar	32
`enum`	integer	arithmetic	scalar	s.d.
`float`	float	arithmetic	scalar	(32)
`double`	float	arithmetic	scalar	s.w.u.
`long double`	float	arithmetic	scalar	s.w.u.
`T *` (*pointer to T*)		pointer	scalar	
`T [...]` (*array of T*)		array	aggregate	
`struct {...}`		struct	aggregate	
`union {...}`		union	aggregate	
`T (...)` (*function returning T*)			function	
`void`			void	

ISO-C verlangt binäre Codierung der integralen Typen. Für die Wertebereiche aller arithmetischen Typen sind Mindestwerte[5] und Größenverhältnisse[6] festgelegt. Die implementierten Größen dieser Datentypen sind in `limits.h` und `float.h` definiert.
In obiger Tabelle bezeichnet `T*` einen Zeiger auf den Typ `T`, `T[...]` ein Array vom Typ `T`, `T(...)` eine Funktion mit Rückgabetyp `T`. `void` ist der leere Typ. Als Rückgabetyp einer Funktion deklariert zeigt er an, daß die Funktion nichts zurückgibt, in der Parameterliste, daß sie nichts nimmt. Ein Zeiger auf `void` ist ein Zeiger auf irgendetwas unbestimmtes, ein generischer Zeiger, den man nie dereferenzieren kann. Variablen oder Arrays vom Typ `void` können daher nicht deklariert werden. Der

[5] z.B. Genauigkeit für float mindestens 6, für double mindestens 10 Dezimalstellen.

[6] Ganzzahltypen: long ≥ int ≥ short ≥ char. Gleitpunkttypen: long double ≥ double ≥ float.

Array-Typ `T[]` und der Funktionstyp `T()` können nicht Typ einer Funktion sein.

Die Gruppen, Klassen und Kategorien dienen zur Kenntlichmachung der auf diesen Typen und in Verbindung mit diesen Typen erlaubten Operationen.

Datentypen können durch die sog. *type qualifiers* `const` und `volatile` weiter qualifiziert werden. Dabei bedeutet `const`, daß ein so bezeichneter Datentyp nur gelesen werden darf (*read only*), d.h. er könnte z.B. in einem solchen Speicherbereich oder im ROM abgelegt sein. `volatile` bedeutet, daß die so qualifizierte Größe durch außerhalb des Wirkungsbereichs des Compilers liegende Einflüsse verändert werden könnte, z.B. kann es sich hier um in den Speicherbereich eingeblendete Hardwareregister (sog. *Ports*) handeln. Dies soll den Compiler davon abhalten, gewisse sonst mögliche Optimierungen des Zugriffs auf die entsprechende Variable vorzunehmen. Beide Qualifizierer können auch zusammen auftreten. Hier einige Beispiele:

```
int i; /* i ist als Variable vom Typ int definiert */
const int ic = 4711; /* ic ist als Konstante vom Typ int definiert */
const int *pc; /* pc ist Zeiger auf konstanten int */
int *const cpi = &i; /* cpi ist konstanter Pointer auf int */
const int *const cpc = &ic; /* konstanter Pointer auf konstanten int */
volatile int vi; /* vi kann durch äußeren Einfluß verändert werden */
const volatile int vci; /* vci ist z.B. ein Timerport */
```

Als `const` vereinbarte Variablen dürfen vom Programm nicht verändert werden. Falls man es versucht, gibt es Fehlermeldungen vom Compiler. Falls man dies jedoch durch in C legale Mittel wie Typumwandlung zu umgehen versucht, kann es je nach System auch zu Laufzeitfehlern führen.

7.2 Deklarationen und Definitionen

C ist eine eingeschränkt blockstrukturierte Sprache, d.h. Blöcke sind das strukturelle Gliederungsmittel. Blöcke werden durch die Blockanweisung `{ ... }` erzeugt. Die Einschränkung ist, daß Funktionsdefinitionen (siehe dort) nur außerhalb von Blöcken möglich sind. Blöcke können beliebig geschachtelt werden. Alles, was außerhalb von Blöcken deklariert oder definiert wird, ist global. Alles, was in einem Block deklariert oder definiert wird, ist lokal zu diesem Block und gilt bis zum Verlassen dieses Blocks. Ein in einem Block deklarierter Name kann einen in einer höheren Ebene deklarierten Namen *maskieren*, d.h. der äußere Name wird verdeckt und das damit bezeichnete Objekt ist dort nicht mehr zugreifbar.

Der Compiler bearbeitet (man sagt auch liest) den Quelltext (genauer die vom Präprozessor vorverarbeitete Übersetzungseinheit) Zeile für Zeile, von links nach rechts und von oben nach unten. Bezeichner müssen grundsätzlich erst eingeführt sein, d.h. deklariert und/oder definiert sein, bevor sie benutzt werden können.

Deklarationen machen dem Compiler Bezeichner (Namen) und ihren Typ bekannt. Sie können auch unvollständig sein, d.h. nur den Namen und seine Zugehörigkeit zu einer bestimmten Klasse bekannt machen, ohne wissen zu müssen, wie der Typ nun genau aussieht. Das reicht dann nicht aus, um dafür Speicherplatz zu

reservieren, aber man kann z.B. einen Zeiger auf diesen jetzt noch unvollständigen Typ erzeugen, um ihn dann später, wenn der Typ vollständig bekannt ist, auch zu benutzen. Deklarationen können, abhängig von ihrer Typklasse, auch Definitionen (s.w.u.) sein. Wenn sie global, d.h. außerhalb von Blöcken erfolgen, sind sie standardmäßig auf den Wert Null initialisiert. Innerhalb eines Blocks ist ihr Wert bei ausbleibender Initialisierung undefiniert. Definitionen haben die Form:

Typ Name; oder *Typ Name1* , *Name2* , ... ;

Definitionen weisen den Compiler an, Speicherplatz bereitzustellen und, wenn das angegeben wird, mit einem bestimmten Wert zu initialisieren. Eine Definition ist gleichzeitig auch eine Deklaration. Eine Definition macht den Typ vollständig bekannt und benutzbar, d.h. es wird Speicherplatz dafür reserviert (im Falle von Datentypen) oder auch Code erzeugt (im Falle von Funktionsdefinitionen, siehe dort). Definitionen von Datenobjekten mit Initialisierung haben die Form:

Typ Name = *Wert*; oder *Typ Name1* = *Wert1* , *Name2* = *Wert2* , ... ;

7.3 Speicherklassen, Sichtbarkeit und Bindung

Außerhalb von Blöcken vereinbarte Objekte gehören zur Speicherklasse *static*. Sie sind vom Start des Programms an vorhanden, und sind global, d.h. im ganzen Programm gültig und sichtbar – sie haben *global scope* und externe Bindung (*external linkage*). Wenn sie nicht im Programm auf bestimmte Werte gesetzt sind, werden sie auf den Wert 0 initialisiert (im Gegensatz zur Speicherklasse *auto* s.w.u.).

Durch Angabe des Schlüsselworts `static` kann der Sichtbarkeitsbereich (*scope*) für so vereinbarte Objekte auf die Übersetzungseinheit (Datei) eingeengt werden, das Objekt hat dann interne Bindung (*internal linkage*) und *file scope*.

Deklarationen und Definitionen in Blöcken können nur *vor* allen Anweisungen (siehe dort) stehen, also zu Beginn eines Blocks. Sie sind lokal zu dem Block, in dem sie erscheinen (*block scope*). Die so vereinbarten Objekte haben die Speicherklasse *auto*, d.h. sie existieren nur, solange der Block aktiv ist und werden bei Eintritt in den Block jedesmal wieder neu erzeugt, jedoch ohne definierten Anfangswert. Durch Angabe des Schlüsselworts `static` kann die Speicherklasse auf *static* geändert werden, ohne daß Sichtbarkeit und Bindung davon berührt würden. Sie sind von Beginn an vorhanden und behalten ihren Wert auch nach Velassen des Gültigkeitsbereichs.

Vereinbarungen von Objekten mittels `register` sind nur in Blöcken oder in Parameterlisten von Funktionen erlaubt und dienen lediglich als Hinweis an den Compiler, er möge sie in (schnellen) Prozessorregistern ablegen. Ob das denn auch geschieht, bleibt dem Compiler überlassen. Auf so vereinbarte Objekte darf der Adreßoperator `&` nicht angewandt weden.

Der Sichtbarkeitsbereich einer Marke (*label*) ist die Funktion, in der sie deklariert ist (*function scope*). (Siehe Abschnitt 7.6, sowie Abschnitt 7.7, Seiten 21 und 23.)

Innerhalb einer Funktion weist man bei Vereinbarung eines Namens mit `extern` darauf hin, daß das Objekt anderweitig definiert ist. Außerhalb von Funktionen gelten alle vereinbarten Objekte defaultmäßig als *extern*.

7.4 Operatoren

C verfügt über einen reichhaltigen Satz von Operatoren. Diese lassen sich nach verschiedenen Kategorien gliedern:

- nach der Art: unäre, binäre und ternäre Operatoren
- nach Vorrang – Präzedenz (*precedence*)
- nach Gruppierung – Assoziativität: links, rechts (*associativity*)
- nach Stellung: Präfix, Infix, Postfix
- nach Darstellung: einfach, zusammengesetzt

C-Bezeichnung	Erläuterung d. Funktion	Klasse	Vorrang	Gruppierung
`[]`	Indexoperator	postfix	16	links
`()`	Funktionsaufruf	postfix	16	links
`.`	direkte Komponentenwahl	postfix	16	links
`->`	indir. Komponentenwahl	postfix	16	links
`++ --`	Postinkrement, -dekrement	postfix	16	links
`++ --`	Präinkrement, -dekrement	präfix	15	rechts
`sizeof`	Größe ermitteln	unär	15	rechts
`~`	bitweise Negation	unär	15	rechts
`!`	logische Negation	unär	15	rechts
`- +`	arithm. Negation, plus	unär	15	rechts
`&`	Adresse von	unär	15	rechts
`*`	Indirektion	unär	15	rechts
(*type name*)	Typumwandlung (*cast*)	unär	14	rechts
`* / %`	mult., div., mod.	binär	13	links
`+ -`	Addition, Subtraktion	binär	12	links
`<< >>`	bitweise schieben	binär	11	links
`< > <= >=`	Vergleiche	binär	10	links
`== !=`	gleich, ungleich	binär	9	links
`&`	bitweises AND	binär	8	links
`^`	bitweises XOR	binär	7	links
`|`	bitweises OR	binär	6	links
`&&`	logisches AND	binär	5	links
`||`	logisches OR	binär	4	links
`? :`	Bedingungsoperator	ternär	3	rechts
`=`	Zuweisung	binär	2	rechts
`+= -= *= /= %=`	Verbundzuweisung	binär	2	rechts
`<<= >>= &= ^= |=`	Verbundzuweisung	binär	2	rechts
`,`	Sequenzoperator	binär	1	links

Die Vielfalt und oft mehrfache Ausnutzung der Operatorzeichen auch in anderem syntaktischen Zusammenhang[7] bietet anfangs ein verwirrendes Bild. Der Compiler kann aber immer nach dem Kontext entscheiden, welche der Operatorfunktionen gerade gemeint ist. Zur Erleichterung des Verständnisses der Benutzung und Funktionsweise der Operatoren daher folgend einige Anmerkungen zur Operatortabelle.

`[]` – siehe Abschnitt 7.9, auf Seite 28.

`()` – siehe Abschnitt 7.8, auf Seite 24.

. und `->` – siehe Abschnitt 7.10, auf Seite 32.

`++`, `--` – (z.B. `a++`, `b--`) – als Postin- bzw. -dekrement liefern sie den ursprünglichen Wert ihres Operanden und erhöhen bzw. erniedrigen den Wert des Operanden *danach* um 1. Diese Operatoren können nur auf Objekte im Speicher angewandt werden, die vom skalaren Typ sein müssen und auf die schreibend zugegriffen werden kann. Wann die tatsächliche Veränderung des Operandenwertes im Speicher eintritt, der *Seiteneffekt* dieser Operatoren, ist implementationsabhängig und erst nach dem Passieren eines *Sequenzpunktes* sicher.

`++`, `--` – (z.B. `++a`, `--b`) – als Präin- bzw. -dekrement erhöhen bzw. erniedrigen sie *erst* den Wert ihres Operanden um 1 und liefern dann den neuen, so erhöhten bzw. erniedrigten Wert. Diese Operatoren können nur auf Objekte im Speicher angewandt werden, die vom skalaren Typ sein müssen und auf die schreibend zugegriffen werden kann. Wann die tatsächliche Veränderung des Operandenwertes im Speicher eintritt, der *Seiteneffekt* dieser Operatoren, ist implementationsabhängig und erst nach dem Passieren eines *Sequenzpunktes* sicher.

`sizeof` – dieser Operator arbeitet zur Compilierungszeit (sog. *compile time operator*) und liefert die Größe seines Operanden in Einheiten des Typs `char`: `sizeof(char) == 1`. Der Operand kann ein Typ sein, dann muß er in `()` stehen, oder ein Objekt im Speicher, dann sind keine Klammern erforderlich. Ist der Operand ein Arrayname, liefert er die Größe des Arrays in `char`-Einheiten.

`~` – (z.B. `~a`) – die Tilde liefert den Wert der bitweisen Negation (das Komplement) der Bitbelegung ihres Operanden, der vom integralen Typ sein muß.

`!` – (z.B. `!a`) – liefert die logische Negation des Wertes seines Operanden, der vom skalaren Typ sein muß. War der Wert 0, ist das Ergebnis 1, war der Wert ungleich 0 , ist das Ergebnis 0.

`-`, `+` – (z.B. `-a`) – die unäre Negation liefert den negierten Wert ihres Operanden, der vom arithmetischen Typ sein muß. Das unäre Plus wurde nur aus Symmetriegründen eingeführt, und dient evtl. lediglich Dokumentationszwecken.

`&` – (z.B. `&a`) – liefert die Adresse eines Objektes im Speicher (und erzeugt somit einen Zeigerausdruck, siehe dazu auch den Abschnitt 7.9, Seite 29).

`*` – (z.B. `*a`) – in einer Deklaration erzeugt dieser Operator einen Zeiger auf den deklarierten Typ, in der Anwendung auf einen Zeigerwert, liefert er den Wert des so bezeigten Objekts (siehe auch im Abschnitt 7.9, Seite 28).

[7] Dann nicht als Operatoren, sondern z.B. als Listentrenner (Komma), oder als Gruppierungsmittel (runde Klammern).

(*typename*) – *typename* ist ein Typbezeichner. Der sog. *type cast operator* liefert den in diesen Typ konvertierten Wert seines Operanden. Dabei wird versucht, den Wert zu erhalten. Eine (unvermeidbare, beabsichtigte) Wertänderung tritt ein, wenn der Wert des Operanden im Zieltyp nicht darstellbar ist, ähnlich einer Zuweisung an ein Objekt dieses Typs. Im Folgenden einige Hinweise zu erlaubten Konversionen:
– Jeder arithmetische Typ in jeden arithmetischen Typ.
– Jeder Zeiger auf `void` in jeden Objektzeigertyp.
– Jeder Objektzeigertyp in Zeiger auf `void`.
– Jeder Zeiger auf ein Objekt oder `void` in einen Integertyp.
– Jeder Integertyp in einen Zeiger auf ein Objekt oder `void`.
– Jeder Funktionszeiger in einen anderen Funktionszeiger.
– Jeder Funktionszeiger in einen Integertyp.
– Jeder Integertyp in einen Funktionszeiger.
Die Zuweisung von `void`-Zeiger an Objektzeiger und umgekehrt geht übrigens auch ohne den Typkonversionsoperator. In allen anderen Fällen ist seine Anwendung geboten oder erforderlich, und sei es nur, um den Warnungen des Compilers zu entgehen. Ob die so konvertierten Objekte dann auch benutzbar sind, und welche (beabsichtigte) Wirkung diese Nutzung dann im Programm haben wird, ist meist implementationsabhängig, und kann, besonders wenn der Wert oder die vorgeschriebene Ausrichtung der Zeiger nicht stimmen, zu Laufzeitfehlern führen. Es ist allerdings garantiert, daß bei der Umwandlung eines beliebigen Objektzeigers in einen `void`-Zeiger und wieder zurück kein Verlust entsteht und die volle Funktionalität des Objektzeigers erhalten bleibt. Die Anwendung dieser Eigenschaft sieht man deshalb häufig bei Funktionen der Standardbibliothek, die `void`-Zeiger als Rückgabetyp oder als Parameter haben. Diese Funktionen wandeln diese Zeiger intern natürlich entsprechend um. Siehe dazu Abschnitt A.2, Seite 38 ff.

`%` – (z.B. `a%b`) – modulo liefert den ganzzahligen Divisionsrest des Wertes seines linken Operanden geteilt durch den Wert seines rechten Operanden und läßt sich nur auf integrale Typen anwenden. Dabei sollte man Überlauf und die Division durch Null vermeiden. Bei positiven Operanden wird der Quotient nach 0 abgeschnitten. Falls negative Operanden beteiligt sind, ist das Ergebnis implementationsabhängig. Es gilt jedoch immer: $X = (X/Y) * Y + (X\%Y)$.

Die übrigen arithmetischen Binäroperatoren können nur auf Operandenpaare vom arithmetischen Typ angewandt werden, dabei geht, wie üblich und auch aus der Tabelle zu ersehen, Punktrechnung vor Strichrechnung. Bei der Ganzzahldivision wird ein positiver Quotient nach 0 abgeschnitten. Man vermeide auch hier die Null als Divisor. Wenn unterschiedliche Typen an der Operation beteiligt sind, wird sebsttätig in den größeren der beteiligten Typen umgewandelt (balanciert).

`<<`, `>>` – (z.B. `a<<b`) – die Bitschiebeoperatoren schieben den Wert des linken Operanden um Bitpositionen des Wertes des rechten Operanden jeweils nach links bzw. rechts und können nur auf integrale Operandenpaare angewandt werden. Für eine n-Bit-Darstellung des promovierten linken Operanden muß der Wert des rechten Operanden im Intervall 0..n-1 liegen. Bei positivem linken Operanden werden

Nullen in die freigewordenen Positionen nachgeschoben. Ob bei negativem linken Operanden beim Rechtsschieben das Vorzeichen nachgeschoben wird (meist so gehandhabt), oder Nullen, ist implementationsabhängig.

Die Vergleichsoperatoren (z.B. `a==b`) können nur auf arithmetische und auf Paare von Zeigern gleichen Typs angewandt werden. Sie liefern den Wert 1, wenn der Vergleich erfolgreich war, sonst 0.

Die bitlogischen Operatoren (z.B. `a&b`) können nur auf integrale Typen angewandt werden und liefern den Wert der bitlogischen Veknüpfung des Wertes des linken mit dem Wert des rechten Operanden (beide als Bitmuster interpretiert).

`&&` – (z.B. `a&&b`) – testet, ob beide Operanden ungleich Null (wahr) sind. Ist der linke Operand wahr, wird auch der rechte getestet, andernfalls hört man auf, und der rechte Operand wird nicht mehr bewertet, da das Ergebnis der logischen UND-Verknüpfung ja schon feststeht (sog. Kurzschlußbewertung, *short circuit evaluation*, mit Sequenzpunkt nach dem linken Operanden). Beide Operanden müssen vom skalaren Typ sein. Im Wahrheitsfall ist der Wert des Ausdrucks 1, sonst 0.

`||` – (z.B. `a||b`) – testet, ob mindestens einer der beiden Operanden ungleich Null (wahr) ist. Ist der linke Operand gleich Null (falsch), wird auch der rechte getestet, andernfalls hört man auf, und der rechte Operand wird nicht mehr bewertet, da das Ergebnis der logischen ODER-Verknüpfung ja schon feststeht (sog. Kurzschlußbewertung, *short circuit evaluation*, wie oben) Beide Operanden müssen vom skalaren Typ sein. Im Wahrheitsfall ist der Wert des Ausdrucks 1, sonst 0.

X`?`Y`:`Z – X muß vom skalaren Typ sein und wird bewertet. Ist X ungleich Null (wahr), wird Y bewertet, andernfalls wird Z bewertet. Y und Z können fast beliebige Ausdrücke sein, auch void ist möglich, sollten aber kompatibel[8] sein. Zwischen der Bewertung von X und der Bewertung von entweder Y oder Z befindet sich ein Sequenzpunkt (*sequence point*). Der Wert des Ausdrucks ist dann der Wert des (evtl. im Typ balancierten) Wertes des zuletzt bewerteten Ausdrucks. Der Nutzen des Bedingungsoperators gegenüber z.B. der `if-else`-Kontrollanweisung, besteht darin, daß er einen Ausdruck erzeugt und deshalb überall dort stehen kann, wo ein Ausdruck gebraucht wird und wo keine Anweisung stehen kann. Er assoziiert von rechts nach links, man kann ihn auch schachteln, sollte das aber tunlichst unterlassen, da dies auch bei guter Klammerung und entsprechender Formatierung schnell unübersichtlich[9] wird. Auch das früher manchmal gebrauchte Argument, daß der Compiler aus einem Bedingungsausdruck effizienteren Code generieren könne, hat keine Bedeutung[10] mehr. Der Bedingungsoperator hat sicher seine Anwendungen, aber die sind, allgemein betrachtet, doch äußerst selten.

[8] Die genaue Erklärung, wann diese Bedingung erfüllt ist, ist ziemlich komplex, wenig praxisrelevant und würde den Rahmen dieser Kurzeinführung, in der es ja nicht darum geht, die Nischen und dunklen Seiten einer Sprache auszuleuchten, bei weitem sprengen. Deshalb sei hier bewußt auf weitere Erläuterungen verzichtet und stattdessen auf die Literatur verwiesen.

[9] Daher ist diese Schachtelung selbst bei hartgesottenen C-Programmierern verpönt!

[10] Wenn es überhaupt je welche hatte! Die Codegenerierung ist ja typischerweise sehr abhängig vom jeweils zur Übersetzung benutzten Compiler.

`=` – Der Zuweisungsoperator bewertet seine beiden Operanden von rechts nach links, so sind auch Zuweisungsketten in der Art von `a = b = c = d = 4711` möglich. Der Wert des Zuweisungsausdrucks ist der Wert des Zugewiesenen, der in den Typ des linken Operanden transformierte Wert des rechten Operanden. Der linke Operand muß ein Objekt im Speicher darstellen, auf das schreibend zugegriffen werden kann. Aufgrund der speziellen Eigenheit von C, daß die Zuweisung ein Ausdruck und keine Anweisung ist, sowie seiner einfachen Wahr-Falsch-Logik, taucht die Zuweisung oft als Testausdruck zur Schleifenkontrolle auf. Ein markanntes Beispiel:

```
while (*s++ = *t++) ; /* C-Idiom für Zeichenkettenkopie */
```

Diese Art ist für die Sprache C idiomatisch, und sollte auch vom Anfänger schnellstens erlernt und beherrscht werden. Wer von anderen Sprachen kommt, sollte darauf achten, daß `=` der Zuweiser ist und `==` der Vergleichsoperator, und es vermeiden, die beiden zu verwechseln. Hilfreiche Compiler warnen hier manchmal.

Die Verbund- oder Kombinationszuweiser bestehen aus zwei Zeichen, deren rechtes der Zuweiser ist. Sie führen, kombiniert mit der Zuweisung verschiedene arithmetische, bitschiebende und bitlogische Operationen aus. Dabei bedeutet `a` *op*`= b` soviel wie `a = a` *op* `b`, mit dem Unterschied, daß `a`, also der linke Operand, nur einmal bewertet wird. Das kann, abgesehen von der größeren Übersichtlichkeit des Ausdrucks, auch beim Hinschreiben helfen, Fehler zu vermeiden (man denke an lange und komplexe Variablennamen!). Der linke Operand muß, wie beim einfachen Zuweiser, immer ein Objekt im Speicher darstellen, auf das schreibend zugegriffen werden kann.

Der Komma- oder Sequenzoperator (z.B. `a,b`) gruppiert wieder von links nach rechts und bewertet erst seinen linken, dann seinen rechten Operanden. Dazwischen liegt ein Sequenzpunkt, das heißt, alle Seiteneffekte sind garantiert eingetreten. Der Wert des Ausdrucks ist das Resultat der Bewertung des rechten Operanden. Der Nutzen des Operators besteht darin, daß er einen Ausdruck erzeugt und folglich überall stehen kann, wo ein Ausdruck gebraucht wird. Seine Hauptanwendung sind die Initialisierungs- und Reinitialisierungsausdrücke in der Kontrollstruktur der `for`-Schleife, wo ja jeweils nur ein Ausdruck erlaubt ist, und manchmal mehrere gebraucht werden. Man vermeide, den Sequenzoperator aus Bequemlichkeit einzusetzen, um z.B. einfach mehrere Ausdrucksanweisungen zu bündeln, da dies bei der Wartung von Quelltexten leicht zu Fehlern führen kann. Die Kommata in Deklarations- oder Parameterlisten sind keine Sequenzoperatoren. Der Sequenzoperator hat absichtlich den allerniedrigsten Vorrang.

Einige Operationen erzeugen implementationsabhängige Typen, die in `stddef.h`[11] definiert sind. `size_t` ist der vom `sizeof`-Operator erzeugte vorzeichenlose integrale Typ. `ptrdiff_t` ist der vorzeichenbehaftete integrale Typ, der vom Subtraktionsoperator erzeugt wird, wenn dieser auf Zeiger (gleichen Typs!) angewandt wird. (Siehe dazu auch Seite 38 oben und Abschnitt 7.9, Seite 29.)

[11] Für die Anwendung dieser Operationen ist die Einbindung dieses Headers nicht erforderlich, man braucht ihn nur bei eigenen Deklarationen für diese Typen.

7.5 Ausdrücke

C ist eine ausdrucksorientierte Sprache. Überall in einem C-Quelltext finden wir Ausdrücke. Der Compiler betrachtet die Ausdrücke (man sagt auch er faßt sie an) und bewertet sie. Ein Audruck (*expression*) in C ist:

- eine Konstante (*constant*)
- eine Variable (*variable*)
- ein Funktionsaufruf (*function call*)
- eine beliebige Kombination der obigen 3 Elemente mittels Operatoren

Jeder Ausdruck hat einen Typ und einen Wert.

Bei der Bewertung von Ausdrücken gelten folgende Regeln: Daten vom Typ `char` oder `short` werden sofort in den Typ `int` umgewandelt (*integral promotion*). Bei der Kombination von Ausdrücken wird balanciert, d.h. der dem Wertebereich oder Speicherplatz nach kleinere Typ wird in den beteiligten, dem Wertebereich oder Speicherplatz nach größeren Typ umgewandelt. Dabei wird versucht, den Wert zu erhalten (*value preservation*)[12].

Die Bewertung der einzelnen Elemente von Ausdrücken folgt Vorrang und Assoziativität der Operatoren. Bei Gleichheit in diesen Eigenschaften ist die Reihenfolge der Bewertung (*order of evaluation*) gleichwohl bis auf wenige Ausnahmen undefiniert, denn der Compiler darf sie auf für ihn »günstige« Weise ändern, wenn das Ergebnis aufgrund der üblichen mathematischen Regeln gleichwertig wäre. In der Theorie gilt $(a \times b)/c = (a/c) \times b$, also darf der Compiler das nach seinem Gusto umordnen[13], und auch Gruppierungsklammern können ihn nicht daran hindern. Das kann aber bei den darstellungsbegrenzten Datentypen im Computer schon zu unerwünschtem Überlauf etc. führen. Deshalb sind in der Sprache C sog. Sequenzpunkte (*sequence point*) definiert. Nach einem Sequenzpunkt kann man sicher sein, daß alle in der Sprache vorgesehenen Wirkungen eingetreten sind. Wichtig ist dies vor allem bei Betrachtung sog. Seiteneffekte. Das Semikolon einer Anweisung, das Komma als Sequenzoperator, der Bedingungsoperator, die logischen Operatoren `&&` und `||` und Funktionsaufrufe enthalten solche Sequenzpunkte.

Manche Operatoren bewirken sog. Seiteneffekte (*side effects*), d.h. sie können den Zustand des Rechners verändern, z.B. den Wert von Speichervariablen oder Registern oder sonstiger Peripherie. Dazu gehören neben den Zuweisern auch die Post- und Präinkrement und -dekrement-Operatoren und Funktionsaufrufe. Das Eintreten der Wirkung dieser Seiteneffekte sollte *niemals* abhängig von der Reihenfolge

[12] in Gegensatz zum alten K&R C, das versuchte, das Vorzeichen zu erhalten (*sign preservation*).

[13] Einzig bei Beteiligung von Gleitpunkttypen ist es dem Compiler untersagt, nach mathematischen Möglichkeiten umzugruppieren. Dies ist wohl vor allem dem Wissen um die Unvollkommenheit der Gleitpunktdarstellung als Abbild der reellen Zahlen geschuldet.

der Bewertung sein! Während kommaseparierte Deklarations- und Definitionslisten strikt von links nach rechts abgearbeitet und bewertet werden, gilt das z.B. für die Reihenfolge der Bewertung in Parameterlisten beim Funktionsaufruf nicht!

7.6 Anweisungen

In C gibt es folgende Anweisungen (*statements*):

- Leeranweisung `;` (*empty statement*)
- Audrucksanweisung *expression* `;` (*expression statement*)
- Blockanweisung `{ ... }` (*block statement*)
- markierte Anweisung *label* : *statement* (*labeled statement*)
- Auswahlanweisung `if else switch ... case` (*selection statement*)
- Wiederholungsanweisung `for while do ... while` (*iteration statement*)
- Sprunganweisung `goto break continue return` (*jump statement*)

7.7 Kontrollstrukturen

Kontrollstrukturen definieren den Ablauf eines Programms. Die einfachste Kontrollstruktur ist die Sequenz, d.h. Folge. Der Compiler liest den Quelltext von links nach rechts, von oben nach unten, und setzt ihn in Code um, der eine sequentielle Abarbeitung bewirkt. Um dies zu erzeugen, schreibt man also eine Anweisung nach der andern, von links nach rechts, bzw. besser von oben nach unten, hin.

Die nächste Kontrollstruktur ist die Auswahl. C kennt zwei Auswahl- oder Verzweigungskontrollstrukturen: `if else`$_{opt}$ und `switch` ... `case`. Das `if`-Konstrukt hat folgende allgemeine Form:

```
if (expression)   /* expression muß vom arithmetischen oder Zeigertyp sein */
      statement1  /* wenn expression ungleich 0, statement1 ausführen */
else
      statement2  /* sonst statement2 ausführen */
```

Der `else`-Teil ist optional. Man beachte, daß es in C kein *then* und keine Endmarke (*endif*, *fi* o.ä.) für diese Konstruktion gibt. Ebenso, und da hilft auch die schönste Einrückung nicht, ist jeweils nur **ein** *statement* erlaubt; braucht man mehrere, so muß man zum `{`*block statement*`}` greifen. Falls man mehrere geschachtelte `if`-Strukturen verwendet, ordnet der Compiler das `else` immer dem jeweilig direkt vorausgehenden `if` zu, sodaß man durch Verwendung von Blockklammern `{}` für die korrekte Gliederung sorgen muß, die visuelle Gestaltung des Quelltexts ist nur eine Lesehilfe für den menschlichen Leser und hat, wie anderer Leerraum auch, für die Syntax der Sprache C keine Bedeutung.

Das zweite Auswahlkonstrukt, `switch` ... `case`, hat viele Formen, am häufigsten gebraucht wird die folgende allgemeine Form:

```
switch (integral expression) {
    case constintexpr1 :    /* Der : ist die Syntaxkennung für eine Marke. */
        statement1
        statement2
        break;              /* hier wird der switch in diesem Fall verlassen. */
    case constintexpr2 :
        statement3
        statement4          /* break fehlt: Es geht weiter zum nächsten Fall! */
    default:
        statement5
}
```

Die Ausdrücke in den `case`-Marken müssen konstante integrale Ausdrücke sein. Mehrere Marken sind erlaubt. Für den kontrollierenden Ausdruck findet Integer-Erweiterung statt und die `case`-Konstanten werden in den so erweiterten Typ umgewandelt. Danach dürfen keine zwei Konstanten den gleichen Wert haben. Die `default`-Marke darf pro `switch` nur einmal vorhanden sein; sie deckt alles ab, was von den anderen Marken nicht erfaßt wird und darf an belieber Stelle erscheinen. Das Problem des `switch`-Konstrukts ist die `break`-Anweisung: fehlt sie, geht die Abarbeitung über die nächste Marke hinweg einfach weiter (sog. *fall through*). Dies kann man natürlich geschickt ausnutzen, ein fehlendes – *vergessenes* – `break` hat jedoch oft schon zu den seltsamsten Überraschungen geführt. Es ist daher zu empfehlen, einen beabsichtigten Fall von *fall through* durch entspr. Kommentar besonders kenntlich zu machen.

Die nächste wichtige Kontrollstruktur ist die Wiederholung, auch Schleife genannt: Hier hält C drei verschiedene Konstrukte bereit:

```
while (expression)   /*solange expression ungleich 0 */
    statement        /* statement ausführen */
```

expression muß vom arithmetischen oder Zeigertyp sein und wird bewertet. Falls nicht 0, wird *statement* ausgeführt; dies wird solange wiederholt, bis *expression* zu 0 bewertet wird. Dies ist eine sog. kopfgesteuerte Schleife. Soll das `while`-Konstrukt mehrere Anweisungen kontrollieren, greift man üblicherweise zur Blockanweisung.

```
do
    statement           /* statement ausführen */
while (expression) ;    /* solange bis expression zu 0 bewertet wird */
```

statement wird ausgeführt, dann wird *expression* bewertet. *expression* muß wie oben vom arithmetischen oder Zeigertyp sein. Falls nicht 0, wird dies solange wiederholt, bis *expression* zu 0 bewertet wird. Man beachte das syntaktisch notwendige Semikolon am Schluß des Konstrukts. Dies ist eine sog. fußgesteuerte Schleife. Für mehrere zu kontrollierende Anweisungen gilt das gleiche wie oben. Das `do ... while`-Konstrukt ist etwas unübersichtlich, weil der kontrollierende Ausdruck am Ende steht. Es wird daher auch nur selten benutzt, und sollte *tunlichst* gemieden werden.

`for` (*expression*1$_{opt}$; *expression*2$_{opt}$; *expression*3$_{opt}$)
statement

Jeder der drei Ausdrücke in der Klammer des `for`-Konstrukts darf auch fehlen, die beiden Semikola sind jedoch syntaktisch notwendig. Zu Beginn wird einmalig *expression*1 bewertet, ihr Typ unterliegt keiner Einschränkung. Sind mehrere Ausdrücke erforderlich, ist dies der Platz für den Einsatz des Sequenzoperators (,). Hier erfolgt daher meist die Initialisierung der Schleife. Als nächstes wird *expression*2 bewertet, sie muß vom arithmetischen oder Zeigertyp sein. Ist der Wert ungleich 0, so wird *statement* ausgeführt. Alsdann wird *expression*3 bewertet, ihr Typ unterliegt keiner Einschränkung. Hier erfolgt meist die Reinitialisierung der Schleife. Dann wird wieder *expression*2 bewertet. Der Zyklus wird solange wiederholt, bis die Bewertung von *expression*2 0 ergibt. Fehlt *expression*2, wird dieser Fall als ungleich 0 bewertet.

Die `for`-Schleife ist gut lesbar und übersichtlich, da Initialisierung, Test und Reinitialisierung dicht beieinander und sofort im Blickfeld sind, bevor man überhaupt mit der Betrachtung des Schleifenkörpers beginnt. Sie ist daher sehr beliebt und mit geschätzten über 70% auch das bei weitem häufigste Schleifenkonstrukt. Es folgt die `while`-Schleife mit geschätzten über 25%. Bedingt durch die Eigenheiten der C-Syntax wird die Hauptarbeit der `for`-Schleife oft im Testausdruck (*expression*2) geleistet, dann wird - syntaktisch notwendig - eine Leeranweisung kontrolliert. Sog. Endlosschleifen formuliert man in C folgendermaßen:

`for (;;)` *statement* oder `while (1)` *statement*

Die Formulierung mit `for`

`for` (*expression*1$_{opt}$; *expression*2$_{opt}$; *expression*3$_{opt}$)
statement

ist äquivalent zu dieser Formulierung mit `while`

expression1 ;
`while` (*expression2*) `{`
statement
*expression*3 ;
`}`

sofern *statement* (z.B. als Blockanweisung) kein `continue` enthält.

Den letzten Teil der Kontrollstrukturen bilden die sog. Sprunganweisungen:

`goto` *label*; springt zu einer Marke in der umgebenden Funktion. Diese Anweisung findet in der strukturierten Programmierung keine Verwendung und wird auch im Systembereich nur selten gebraucht. Sie ist jedoch nützlich in der (nicht für menschliche Leser bestimmten) maschinellen Codegenerierung.

`break` ; darf nur in `switch` oder in Wiederholungsanweisungen stehen und bricht aus der es umgebenden Anweisung aus.

`continue` ; darf nur in Wiederholungsanweisungen stehen und setzt die es umgebende Anweisung am Punkte der Wiederholung fort.

`return` *expression*$_{opt}$; kehrt aus einer umgebenden Funktion mit der optionalen *expression* als Rückgabewert zurück.

7.8 Funktionen

Funktionen sind das Hauptgliederungsmittel eines Programms. Jedes gültige C-Programm muß eine bestimmte Funktion enthalten, nämlich die Funktion `main()`. Funktionen in C erfüllen die Aufgaben, die in anderen Programmiersprachen *function*, *procedure* oder *subroutine* genannt werden. Sie dienen dazu, die Aufgaben des Programms in kleinere, übersichtliche Einheiten mit klaren und wohldefinierten Schnittstellen zu unterteilen. Funktionsdeklarationen haben die allgemeine Form:

Typ `Funktionsname(`$Parameterliste_{opt}$`);`

Wird *Typ* nicht angegeben, so wird `int` angenommen, man sollte dies aber unbedingt vermeiden (in C99 ist es untersagt!). Ist die Parameterliste leer, kann die Funktion eine unspezifizierte Anzahl (auch Null) Parameter unspezifizierten Typs nehmen. Besteht die Parameterliste nur aus dem Schlüsselwort `void`, nimmt die Funktion keine Parameter. Andernfalls enthält die Parameterliste einen oder mehrere Typnamen, optional gefolgt von Variablennamen, als kommaseparierte Liste. Als letzter (von mindestens zwei) Parametern ist als Besonderheit auch die Elipse (...) erlaubt und bedeutet dann eine variable Anzahl sonst unspezifizierter Parameter. Die Variablennamen haben für den Compiler keine Bedeutung, können aber dem Programmierer als Hinweis auf die beabsichtigte Verwendung dienen, im Sinne einer besseren Dokumentation. Zum Beispiel sind diese beiden Deklarationen,

```
int myfunc(int length, int count, double factor); oder
int myfunc(int, int, double);
```

auch Funktionsprototypen genannt, für den Compiler identisch. Die Typangaben der Parameterliste, ihre Anzahl und Reihenfolge – auch Signatur (*signature*) genannt – dienen dem Compiler zur Fehlerdiagnose beim Aufruf, d.h. der Benutzung der Funktion (s.w.u.). Deshalb sollten Funktionsdeklarationen – die Prototypen – der Benutzung der Funktionen – dem Funktionsaufruf (*function call*) – immer vorausgehen. Funktionsdefinitionen haben die allgemeine Form:

Typ `Funktionsname(`$Parameterliste_{opt}$`)`

```
{
        Deklarationen und Definitionen

        Anweisungen
}
```

Kein Semikolon, weder nach der schließenden Klammer der Parameterliste, noch nach der schließenden Klammer des Blocks! Funktionen können nur außerhalb von Blöcken definiert werden. Eine Funktionsdefinition ist immer auch gleichzeitig eine Deklaration. Der Hauptblock einer Funktion, auch Funktionskörper genannt, ist der einzige Ort, wo Code (im Sinne von ausführbaren Prozessorbefehlen) erzeugt werden kann. Typ und Signatur einer Funktion müssen mit etwaigen vorausgegangenen Prototypen übereinstimmen, sonst gibt es Fehlermeldungen vom Compiler.

Beim Funktionsaufruf (*function call*) schreibt man lediglich den Namen der Funktion, gefolgt von den in Klammern gesetzten *Argumenten*, oft auch *aktuelle*

Parameter genannt, als kommaseparierte Liste. Die Argumente nehmen den Platz der *formalen Parameter* ein und werden, da dies ein Zuweisungskontext ist, im Typ angeglichen. Die Reihenfolge der Bewertung dieser Argumentzuweisung ist dabei nicht festgelegt – es ist nur sichergestellt, daß alle Argumente bewertet sind, bevor der eigentliche Aufruf, d.h. der Sprung zum Code des Funktionskörpers erfolgt. Falls die Funktion ein Ergebnis liefert – den sog. Funktionswert, kann man dieses zuweisen oder sonstwie verarbeiten, muß es aber nicht (wenn man z.B. nur an dem Seiteneffekt interessiert ist). Beispiel:

```
len = strlen("hello, world\n"); /* Funktionswert zuweisen */
printf("hello, world\n"); /* kein Interesse am Funktionswert */
```

Ein Funktionsaufruf stellt einen *Ausdruck* dar und darf überall stehen, wo ein Ausdruck des Funktionstyps stehen kann. Eine `void`-Funktion hat definitionsgemäß keinen Funktionswert und ihr Aufruf darf daher nur in einem für diesen Fall zulässigen Zusammenhang erscheinen (z.B. nicht in Zuweisungen, Tests etc.).

Die Ausführung des Funktionskörpers endet mit einer `return`-Anweisung mit einem entspr. Ausdruck, dies ist wieder als Zuweisungskontext zu betrachten, und es wird in den Typ der Funktion konvertiert. Eine `void`-Funktion endet mit einer ausdruckslosen `return`-Anweisung oder implizit an der endenden Klammer des Funktionsblocks.

Die Funktion main()

Die Funktion `main()` spielt eine besondere Rolle in der Sprache C. Ihre Form ist vom System vordefiniert, sie wird im Programm nicht aufgerufen, denn sie stellt das Programm selbst dar[14]. Die Funktion `main()` wird vom C-*start up code*, der vom Linker dazugebunden wird, aufgerufen, d.h. das Programm beginnt mit der Ausführung der ersten Anweisung im Funktionskörper von `main()`. Die Funktion `main()` hat zwei mögliche Formen – mit oder ohne Parameter:

`int main(void) {` *Körper von main()* `}` oder

`int main(int argc, char *argv[]) {` *Körper von main()* `}`

Die erste Form verwendet man, wenn das Programm keine Parameter nimmt, die zweite, wenn Parameter auf der Kommandozeile übergeben werden, die dann im Programm ausgewertet werden sollen.

Im zweiten Fall – `argc` (*argument count*) und `argv` (*argument vector*) sind hierbei lediglich traditionelle Namen – bedeutet der erste Parameter die Anzahl der Argumente, incl. des Programmnamens selbst, und ist daher immer mindestens 1. Der zweite Parameter, also `argv`, ist ein Zeiger auf ein Array von Zeigern auf nullterminierte C-Strings, die die Kommandozeilenparameter darstellen. Dieses Array ist selbst auch nullterminiert, also ist `argv[argc]`==0, der sog. Nullzeiger. Der erste Parameter, `argv[0]`, zeigt traditionell[15] auf den Namen, unter dem das Programm aufgerufen wurde. Falls dieser nicht zur Verfügung steht, zeigt `argv[0]` auf den

[14] Dafür gibt es in anderen Programmiersprachen manchmal das Schlüsselwort program o.ä.

[15] unter Unix und – davon abgeleitet – auch auf vielen anderen Systemen

Leerstring, "", d.h. `argv[0][0]` ist `'\0'`. Die in `argc` und `argv` gespeicherten Werte, sowie der Inhalt der damit designierten Zeichenketten können vom Programm gelesen und dürfen, wenn gewünscht, auch verändert werden.

Vor Beginn der Ausführung von `main()` sorgt das System dafür, das alle statischen Objekte ihre im Programm vorgesehenen Werte enthalten. Ferner werden zur Interaktion mit der Umgebung drei Dateien geöffnet:

`stdin` *standard input* Standardeingabestrom (meist Tastatur)

`stdout` *standard output* Standardausgabestrom (meist Bildschirm)

`stderr` *standard error* Standardfehlerstrom (meist Bildschirm)

Diese Standardkanäle (in C und Unix wie Dateien behandelt) haben den Datentyp `FILE*` und sind definiert im Header `stdio.h`.

In beiden möglichen Formen ist `main()` als `int`-Funktion spezifiziert, der Wert ist die Rückgabe an das aufrufende System und bedeutet den Exit-Status des Programms, der dann z.B. Erfolg oder Mißerfolg ausdrücken oder sonstwie vom aufrufenden System ausgewertet werden kann.

Das Programm, bzw. `main()`, endet mit der Ausführung einer `return`-Anweisung mit einem entspr. Ausdruck, dies ist ein Zuweisungskontext, und es wird in den Typ von `main()`, d.h. nach `int`[16] konvertiert. `main()` endet auch, wenn irgendwo im Programm, d.h. auch innerhalb einer ganz anderen Funktion, die Funktion `exit()`, definiert in `stdlib.h`, mit einem entspr. Wert aufgerufen wird. Dieser Wert gilt dann als Rückgabewert von `main()`.

Wenn `main()` mit einer ausdruckslosen `return`-Anweisung oder an der schließenden Klammer seines Funktionsblocks endet[17], ist der Rückgabewert unbestimmt[18].

Bei der Beendigung von `main()` werden erst alle mit `atexit()`, ebenfalls definiert in `stdlib.h`, registrierten Funktionen in umgekehrter Reihenfolge ihrer Registrierung aufgerufen. Sodann werden alle geöffneten Dateien geschlossen, alle mit `tmpfile()` (definiert in `stdio.h`) erzeugten temporären Dateien entfernt und schließlich die Kontrolle an den Aufrufer zurückgegeben.

Das Programm kann auch durch ein – von ihm selbst mit der Funktion `raise()` (definiert in `signal.h`), durch einen Laufzeitfehler (z.B. unbeabsichtigte Division durch Null, illegale Speicherreferenz durch fehlerhaften Zeiger, etc.) oder sonst fremderzeugtes – Signal oder durch den Aufruf der Funktion `abort()` (definiert in `stdlib.h`) terminieren. Was dann im einzelnen geschieht, wie und ob geöffnete Dateien geschlossen werden, ob temporäre Dateien entfernt werden und was dann der Rückgabestatus des Programms ist, ist implementationsabhängig.

[16] Wie groß auch immer der int auf dem jeweiligen System sein mag, unter Unix – und vielen anderen Systemen – werden immer nur die niederwertigen 8 Bit als Rückgabewert verwendet

[17] Dies gilt gemeinhin als fehlerhafte Programmierung, auch wenn viele Compiler milde darüber hinwegsehen und höchstens eine Warnung ausgeben!

[18] Für Neugierige: Es ist der – zum Zeitpunkt der Terminierung dort gerade vorhandene – Wert im Standardrückgaberegister des Prozessors!

7.9 Vektoren und Zeiger

Vektoren (meist *Arrays*, deutsch zuweilen auch Felder genannt) sind als Aggregate komplexe Datentypen, die aus einer Anreihung gleicher *Elemente* bestehen. Diese Elemente werden aufeinanderfolgend in Richtung aufsteigender Adressen im Speicher abgelegt, sie können einfache oder selbst auch wieder komplexe Datentypen darstellen. Die Adresse des Arrays ist identisch mit der Adresse des Elements mit der Nummer 0, denn in C werden die Elemente beginnend mit 0 durchnumeriert. Ein Array wird deklariert mit dem Operator[19] `[]`, in dem die Dimensionsangabe, d.h. die Anzahl der Elemente steht. Die angegebene Anzahl muß eine vorzeichenlose integrale Konstante sein, eine Anzahl 0 ist nicht erlaubt. Der Bezeichner für ein Array ist fest mit seinem Typ verbunden, stellt aber kein Objekt im Speicher dar. In Zusammenhang mit dem `sizeof`-Operator wird die Größe des Arrays als Anzahl von Einheiten des Typs `char` geliefert. Der Compiler sorgt, wie auch sonst immer, für eine korrekte Ausrichtung im Speicher. Ein Beispiel:

```
int iv[10]; /* Ein Array iv von zehn Elementen vom Typ int */
```

Mehr Dimensionen sind möglich, im Gegensatz zu einigen anderen Programmiersprachen gibt es jedoch keine echten mehrdimensionalen Arrays sondern nur Arrays von Arrays (mit beliebiger – vielleicht von Implementation oder verfügbarem Speicherplatz begrenzter – Anzahl der Dimensionen). Ein Beispiel für die Deklaration eines zweidimensionalen Arrays:

```
double dvv[5][20]; /* Array von 5 Arrays von je 20 doubles */
```

Die Ablage im Speicher erfolgt hierbei zeilenweise (bei zwei Dimensionen), d.h. der der rechte Index (der Spaltenindex bei zwei Dimensionen) variiert am schnellsten, wenn die Elemente gemäß ihrer Reihenfolge im Speicher angesprochen werden.

Auf die Elemente zugegriffen wird mit dem Operator[20] `[]`, in dem nun der Index steht, für ein Array mit n Elementen reicht der erlaubte Indexbereich von 0 bis n−1. Beispiel:

```
abc = iv[3]; /* Zuweisung des vierten Elements von iv an abc */
xyz = dvv[i][j]; /* Zuweisung aus dvv, i und j sind Laufvariablen */
```

Die Schrittweite des Index ist so bemessen, daß immer das jeweils nächste – oder vorherige – Element erfaßt wird. Es ist erlaubt, negative Indizes oder solche größer als n−1 zu verwenden, was jedoch beim Zugriff außerhalb des erlaubten Bereichs des so indizierten Arrays passiert, ist implementationsabhängig (und daher normalerweise nicht zu empfehlen!).

Zeiger (*pointer*) sind komplexe Datentypen. Sie beinhalten sowohl die Adresse des Typs, auf den sie zeigen, als auch die Eigenschaften eben dieses Typs, insbesondere, wichtig für die mit ihnen verwendete Adreßarithmetik, seine Speichergröße. Zeiger werden deklariert mittels des Operators[21] `*`.

```
int *ip; /* ip ist ein Zeiger auf Typ int */
```

[19] in diesem Kontext Dimensionsoperator genannt

[20] jetzt Indexoperator genannt

[21] in diesem Zusammenhang Referenzierungs- oder Zeigeroperator genannt

Zeiger müssen initialisiert werden, bevor sie zum Zugriff auf die mit ihnen bezeigten Objekte benutzt werden können:

```
ip = &abc; /* ip wird auf die Adresse der int-Variablen abc gesetzt */
```

Zeiger können nur mit Zeigern gleichen Typs initialisiert werden, oder mit Zeigern auf `void`, (also auf nichts bestimmtes). Zum Initialisieren von Zeigern wird meist der Adreßoperator `&` verwendet, der einen Zeiger auf seinen Operanden erzeugt. In einem Zuweisungszusammenhang gilt der Name eines Arrays als Zeiger auf das erste Element (das mit dem Index 0) des Arrays, d.h. wenn wie im Beispiel weiter oben iv ein Array vom Typ int ist:

```
ip = iv; /* gleichwertig mit ip = &iv[0] */
```

Der Zugriff auf das vom Zeiger referenzierte Objekt, (die sog. Dereferenzierung), geschieht mittels des Operators[22] `*`:

```
if (*ip) .../* testen des Inhalts der Variablen, auf die ip zeigt ... */
```

Wenn `ip` auf `iv` zeigt, dann ist `*ip` identisch mit `iv[0]`, man hätte auch schreiben können `ip[0]` oder `*iv`.

Hier zeigt sich nun der grundlegende Zusammenhang zwischen Array- und Zeigernotation in der Sprache C, es gilt:

`a[n]` ist identisch mit `*(a+n)`

Zu beachten ist hierbei lediglich, daß Arraynamen in einem Zugriffskontext feste Zeiger sind (Adressen), sie stellen kein Objekt im Speicher dar und können somit auch nicht verändert werden, wohingegen Zeigervariablen Objekte sind.

Ein Zeiger kann inkrementiert und dekrementiert werden, d.h. integrale Größen können addiert oder subtrahiert werden, der Zeiger zeigt dann auf ein dem Vielfachen seiner Schrittweite entsprechend entferntes Objekt.

Zeiger gleichen Typs dürfen miteinander verglichen oder voneinander subtrahiert werden. Wenn sie in den gleichen Bereich (z.B. ein entspr. deklariertes Array) zeigen, ergibt sich eine integrale Größe, die den Indexabstand der so bezeigten Elemente bedeutet. Wenn das nicht der Fall ist, ist diese Operation nicht sinnvoll. Erlaubt (und häufig angewandt) ist auch das Testen des Wertes eines Zeigers.

Die Zuweisung integraler Werte an einen Zeiger hat die Bedeutung einer Adresse des vom Zeiger bezeigten Typs. Wenn die Bedingungen der Ausrichtung dieses Typs (z.B. ganzzahlige Vielfache einer best. Größe) nicht erfüllt sind oder der Zugriff auf diese Adresse in der entspr. Typgröße nicht erlaubt sein sollte, kann dies zu Laufzeitfehlern führen. Der Wert 0 eines Zeigers hat die Bedeutung, daß dieser Zeiger ungültig ist, ein sog. Nullzeiger (*null pointer*) – der Zugriff auf die Adresse 0 ist in einem C-System, gleich ob lesend oder schreibend, allgemein nicht gestattet.

Zeiger auf den Typ `void` (also auf nichts bestimmtes) dienen als generische Zeiger lediglich zur Zwischenspeicherung von Zeigern auf Objekte bestimmten Typs. Man kann sonst nichts sinnvolles mit ihnen anfangen, auch keine Adreßberechnungen. Sie dürfen ohne weiteres allen Typen von Zeigern zugewiesen werden und umgekehrt.

[22] jetzt Indirektions-, Dereferenzierungs- oder Inhaltsoperator genannt

Initialisierung von Arrays

Wenn erwünscht, können Arrays durch Angabe einer Initialisierungsliste mit konstanten Werten initialisiert werden, hierbei darf dann die Dimensionsangabe fehlen, man spricht dann von einem unvollständigen Arraytyp (*incomplete array type*), und der Compiler errechnet sie selbsttätig aus der Anzahl der angegebenen Elemente der Liste (und komplettiert damit den Typ!):

```
int magic[] = {4711, 815, 7, 42, 3}; /* magic hat 5 Elemente */
```

Ist die Dimension angegeben, werden die Elemente beginnend mit dem Index 0 mit den Werten aus der Liste initialisiert und der Rest, so vorhanden, wird auf 0 gesetzt:

```
long prim[100] = {2, 3, 5, 7, 11}; /* ab Index 5 alles auf 0 */
```

Die Dimensionsangabe darf nicht geringer als die Anzahl der Elemente der Initialisierungsliste sein:

```
float spec[2] = {1.414, 1.618, 2.718}; /* Fehler! */
```

Die Initialisierung geht auch bei mehr als einer Dimension, hier darf nur die höchste (linke) Dimension fehlen, der Compiler errechnet sie dann:

```
int num[][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}; /* 3 × 3 */
```

Sind alle Dimensionen angegeben und sind weniger Initialisierer da, werden die restlichen Elemente wie gehabt mit 0 initialiert:

```
int num[3][3] = {{1, 2, 3}, {4, 5, 6}}; /* 3 × 3 */
```

Hier – oder im obigen Beispiel – hätte man die inneren geschweiften Klammern auch weglassen können, denn der Compiler füllt bei jeder Dimension beginnend mit dem Index 0 auf, wobei der rechte Index am schnellsten variiert. Oft besteht jedoch die Gefahr der Mehrdeutigkeit und hilfreiche Compiler warnen hier!

Bei der Initialisierung von `char`-Arrays mit konstanten Zeichenketten darf man die geschweiften Klammern weglassen:

```
char mword[] = "Abrakadabra"; /* mword hat 12 Elemente */
```

anstatt:

```
char mword[] = {'A','b','r','a','k','a','d','a','b','r','a','\0'};
```

oder:

```
char mword[] = {"Abrakadabra"};
```

Auch hier zählt der Compiler wieder die Anzahl der Elemente ab (incl. der terminierenden Null) und dimensioniert das Array selbsttätig. Eine evtl. vorhandene Dimensionsangabe muß mindestens der erforderlichen Anzahl entsprechen, überzählige Elemente werden auch hier mit 0 aufgefüllt:

```
char name[64] = "Heiner Mueller"; /* Ab Index 14 alles Nullen */
```

Man beachte folgenden wichtigen Unterschied:

```
char xword[] = "Hokuspokus"; /* xword hat 11 Elemente */
char *xptr = "Hokuspokus"; /* xptr zeigt auf Zeichenkettenkonstante */
```

Im ersten Fall handelt es sich um ein Array namens xword von 11 Elementen in Form eines C-Strings (mit terminierender Null), im zweiten Fall haben wir mit xptr einen Zeiger, der auf einen an anderer Stelle (möglicherweise im Nurlesebereich) gespeicherten C-String (jetzt als namenloses Array vom Typ char) zeigt.

7.10 Strukturen

Eine Struktur (in anderen Sprachen oft als *record*, Verbund, Datensatz bezeichnet) ist als Aggregat ein komplexer Datentyp, der aus einer Anreihung von einer oder mehreren *Komponenten* (*members*) oft auch verschiedenen Typs besteht, um diese so zusammengefaßten Daten dann als Einheit behandeln zu können.

Eine Struktur wird definiert mit dem Schlüsselwort *struct* gefolgt von einem Block mit den Deklarationen der Komponenten. Beispiel:

```
struct person {
    int num;
    char name[64];
    char email[64];
    char telefon[32];
    char level;
};
```

Hier werden mit dem Schlüsselwort `struct` und dem Bezeichner `person`, dem sog. Etikett (*structure tag*), zusammengehörige Daten in einer Struktur zusammengefaßt: Es wird ein neuer, *benutzerdefinierter* Datentyp namens `struct person` geschaffen.

Die Namen der in dem Strukturblock deklarierten Komponenten befinden sich in einem eigenen Namensraum und können nicht mit anderen (äußeren) Namen oder Namen von Komponenten in anderen Strukturen kollidieren. Es wird hierbei auch noch kein Speicherplatz reserviert, sondern lediglich der Typ bekannt gemacht, seine *Form* beschrieben, also ein Bauplan zur Beschaffenheit dieses Typs und seiner Struktur vorgelegt.

Speicherplatz kann reserviert und somit Variablen dieses Typs erzeugt werden, indem man zwischen der beendenden geschweiften Klammer des Strukturblocks und dem abschließenden Semikolon eine Liste von Variablennamen einfügt. Übersichtlicher ist wohl aber meist, die Beschreibung der Form von der Speicherplatzreservierung zu trennen. Variablen dieses Typs werden dann z.B. so vereinbart:

```
struct person hugo, inge, *pp; /* 2 Variablen und ein Zeiger */
```

Man kann natürlich auch gleich ganze Arrays von diesem neuen Typ erzeugen:

```
struct person ap[100]; /* Array von 100 struct person */
```

Der Compiler sorgt dafür, daß die Komponenten der Strukturen in der Reihenfolge ihrer Deklaration mit der korrekten Ausrichtung angelegt werden und daß die Gesamtheit der Struktur so gestaltet ist, daß sich mehrere davon als Elemente eines Arrays anreihen lassen. Je nach Gestalt der Struktur, abhängig von Maschinenarchitektur und Compiler können dabei zwischen den Komponenten und am Ende der Struktur auch Lücken entstehen, sodaß die Gesamtgröße einer Struktur (zu ermitteln mithilfe des `sizeof`-Operators) u.U. größer ist als die Summe der Größen ihrer Komponenten. Der Speicherinhalt der so entstandenen Lücken bleibt dabei undefiniert.

Auf die Komponenten zugegriffen wird direkt mit dem .-Operator[23]:

```
hugo.num = 4711; /* Schreibzugriff auf Komponente num von hugo */
```

Der indirekte Zugriff (über Zeiger) geschieht mithilfe des ->-Operators[24]:

```
pp = &hugo; pp->level = 12; /* Zugriff auf Komponente level von hugo */
```

Oder entsprechend bei Zugriff auf ein Element eines Arrays:

```
ap[5].num = 4712; printf("%d", (ap+5)->num);
```

Strukturen können selbst auch wieder (andere) Strukturen als Komponenten enthalten. Erlaubt ist auch die Definition von Strukturen innerhalb des Strukturdefinitionsblocks – dieser Typ ist dann allerdings auch im Sichtbarkeitsbereich der einbettenden Struktur bekannt, daher sollte dies besser vermieden werden.

Wenn die Definition des Strukturblocks nicht erfolgt oder noch nicht abgeschlossen ist, spricht man von einem unvollständigen (*incomplete*) Datentyp. Davon lassen sich dann zwar keine Variablen erzeugen – Speicherplatzverbrauch und Gestalt sind ja noch unbekannt, es lassen sich aber schon Zeiger auf diesen Typ erstellen. Auf diese Weise können Strukturen Zeiger auf ihren eigenen Typ enthalten, eine Konstruktion, die oft zur Erzeugung von verketteten Listen verwandt wird. Beispiel:

```
struct mlist {
    struct mlist *prev;
    struct mlist *next;
    char descr[64];
};
```

Strukturen können (an Variablen gleichen Typs) zugewiesen werden, als Argumente an Funktionen übergeben und als Rückgabetyp von Funktionen deklariert werden. Die Zuweisung ist dabei als komponentenweise Kopie definiert. Bei größeren Strukturen empfiehlt sich bei den beiden letzteren Aktionen allerdings, lieber mit Zeigern zu arbeiten, da sonst intern immer über tempoäre Kopien gearbeitet wird, was sowohl zeit- wie speicherplatzaufwendig wäre. Strukturvariablen lassen sich ähnlich wie Arrays mit Initialisierungslisten initialisieren.

Syntaktisch ähnlich einer Struktur ist die Variante oder Union (*union*), mit dem Unterschied, daß die verschiedenen Komponenten nicht nacheinander angeordnet sind, sondern alle an der gleichen Adresse liegend abgebildet werden. Vereinbart werden sie mit dem Schlüsselwort `union`, gefolgt von einem optionalen Etikett, gefolgt von einem Definitionsblock mit den Definitionen der Komponenten, gefolgt von einem Semikolon. Sie werden benutzt, um Daten unterschiedlichen Typs am gleichen Speicherplatz unterbringen zu können (natürlich immer nur einen Typ zur gleichen Zeit!), oder um den Speicherplatz anders zu interpretieren (siehe dazu als Beispiel das Programm 2 im Anhang auf Seite 50).

Der Compiler sorgt dafür, daß die Größe der Union, ihre Ausrichtung inklusive etwaiger Auffüllung den Anforderungen der Maschine entsprechen, daher ist die Größe einer Unionsvariablen immer mindestens so groß wie die Größe ihrer größten Komponente.

[23] direkter Komponentenauswahloperator (*direct member selection operator*)

[24] indirekter Komponentenauswahloperator (*indirect member selection operator*)

Bitfelder

Als mögliche Komponenten von *struct* oder *union* können Bitfelder vereinbart werden. Ein Bitfeld dient zur Zusammenfassung von Information auf kleinstem Raum (nur erlaubt innerhalb *struct* oder *union*). Es gibt drei Formen von Bitfeldern:

- normale Bitfelder (*plain bitfields*) – deklariert als *int*
- vorzeichenbehaftete (*signed bitfields*) – deklariert als *signed int*
- nicht vorzeichenbehaftete (*unsigned bitfields*) – deklariert als *unsigned int*

Ein Bitfeld belegt eine gewisse, aufeinander folgende Anzahl von Bit in einem Integer. Es ist nicht möglich, eine größere Anzahl von Bit zu vereinbaren, als in der Speichergröße des Typs *int* Platz haben. Es darf auch unbenannte Bitfelder geben, auf die man dann natürlich nicht zugreifen kann, dies dient meist der Abbildung der Belegung bestimmter Register oder Ports. Hier die Syntax:

```
struct sreg {
    unsigned int
        cf:1, of:1, zf:1, nf:1, ef:1, :3,
        im:3, :2, sb:1, :1, tb:1;
};
```

Nach dem Doppelpunkt steht die Anzahl der Bit, die das Feld belegt. Unter der Annahme, daß der Compiler die Bitfelder vom niederwertigsten Bit aufsteigend in einem 16 Bit großen Integer anordnet, hier z.B. die Modellierung des Statusregisters eines M68000 als Bitfeld, die unbenannten Felder sind reserviert oder sonstwie nicht belegt. Der Zugriff auf die einzelnen Bit geschieht dann auf die bei Strukturen übliche Weise:

```
struct sreg mysreg;
mysreg.cf = 0;
mysreg.im = 07;
mysreg.tb = 1;
```

u.s.w.u.s.f. Wie der Compiler die Bitfelder anlegt, wie er sie ausrichtet und wie groß er die sie enthaltenden Integraltypen macht, ist völlig implementationsabhängig. Wenn man sie überhaupt je verwenden will, wird empfohlen, sie jedenfalls als *unsigned int* zu deklarieren.

Da Bitfelder nur innerhalb von *struct* oder *union* existieren, gibt es keine Vektoren von Bitfeldern, sie haben auch keine Adressen, so daß man den Adreßoperator nicht auf sie anwenden darf. Bitfelder sind so maschinenabhängig, daß niemand sie verwenden sollte[25], sie eignen sich nicht dazu, externe Verhältnisse portabel abzubilden. Außerdem erzeugen die Compiler für ihre Übersetzung meist äußerst platzfressenden und ineffizienten Code!

[25] Man beachte dazu auch die Warnungen bei [Kern89], 6.9, und besonders die Ermahnungen bei [Kern99], pp. 183, 191, 195.

7.11 Aufzählungstypen

Aufzählungstypen – Schlüsselwort *enum* – sind benannte Ganzzahlkonstanten (*enumeration constants*), deren Vereinbarungssyntax der von Strukturen ähnelt. Im Gegensatz zu mit `#define` vereinbarten Konstanten, die ja der C-Präprozessor verarbeitet, werden die *enum*-Konstanten vom C-Compiler selbst bearbeitet.

Sie sind kompatibel zum Typ, den der Compiler dafür wählt – einen Typ, aufwärtskompatibel zum Typ *int*: Es könnte also auch *char* oder *short* sein, aber nicht *long*, das ist implementationsabhängig – und lassen sich ohne weiteres in diesen überführen und umgekehrt, ohne daß der Compiler prüft, ob der Wert auch im passenden Bereich liegt. Hier einige Beispiele zur Deklaration, bzw. Definition:

```
enum color {red, green, blue} mycolor, hercolor;
enum month {JAN=1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC};
enum month mymonth;
enum range {VLO=-10, LLO=-5, LO=-2, ZERO=0, HI=2, LHI=5, VHI=10, OVL};
enum range myrange, hisrange;
enum level {AF=-3, BF, CF, DF, EF, FF, GF, HF} xx, yy, zz;
```

Bei den deklarierten Aufzählungen sind `color`, `month` und `range` wieder das jeweilige Etikett (*tag*), die Namen der für diese Typen definierten Variablen sind entspr. `mycolor`, `mymonth` etc. Die Werte der innerhalb der geschweiften Klammern aufgeführten Konstanten werden, wenn keine Wertzuweisung – die ein ganzzahliger konstanter Ausdruck sein muß – erfolgt, von links beginnend, von Null hochgezählt, ansonsten wird nach jeder Wertzuweisung, wie nach einer Unterbrechung der Zählung, von da an einfach weitergezählt: `red` hat also den Wert 0, `green` den Wert 1 und `blue` den Wert 2, `DEC` ist demnach 12 und `OVL` hat den Wert 11. Dementsprechend hat `DF` den Wert 0, und `HF` repräsentiert den Wert 4.

Bei aller semantischen Nähe zum Typ *int* sind *enum*-Konstanten oft der beste Weg, um mittels *benannter* Konstanten das Programm übersichtlicher zu machen und »magische« Zahlen (*magic numbers*) zu vermeiden, besser oft als die übliche Methode der `#define`-Makros und daher für diesen Zweck sehr zu empfehlen. Diese Art der Verwendung funktioniert natürlich nur für Ganzzahlkonstanten, die den Wertebereich eines `int` nicht überschreiten. Um den Quelltext lesbarer zu gestalten, empfiehlt es sich auch, die Konvention der Großschreibung für Konstanten beizubehalten. *enum*-Konstanten dürfen übrigens auch zur Dimensionierung von Arrays verwandt werden.

Will man keine Variablen eines Aufzählungstyps erzeugen und einfach nur Konstanten definieren, kann man das Etikett auch weglassen:

```
enum {SIZE=1000, DURA=100};        int arr[SIZE], len=DURA;
```

Anders als bei den Bezeichnern innerhalb von *struct* und *union* befinden sich die *enum*-Konstanten auf der gleichen Ebene alle im gleichen Namensraum, man darf also nicht zweimal den gleichen Bezeichner verwenden, auch wenn verschiedene – oder gar keine – Etiketten benutzt werden.

7.12 Typdefinitionen

Das Schlüsselwort ist *typedef*. Der Name läßt es zwar vermuten, aber *typedef* dient nicht zur Definition neuer Datentypen, er erzeugt syntaktisch nur andere Namen (Synonyme, Aliasse) für schon bekannte Typen. Das kann, richtig angewandt, zur erhöhten Lesbarkeit des Quelltextes genutzt werden. Einerseits wird *typedef* dazu benutzt, komplizierte oder umständliche Deklarationen zu vereinfachen, andererseits kann durch geschickten Einsatz die Portabilität von Programmcode auf unterschiedliche Umgebungen erhöht werden. Der so erzeugte »neue« Typ ist mit seinem Ursprungstyp voll kompatibel und syntaktisch quasi-identisch. Die Syntax ist:
`typedef` *bekannter-Typ* *neuer-Typname* `;`. Ein Beispiel:

```
typedef struct mlist {
    struct mlist *prev;
    struct mlist *next;
    char descr[64];
} MLIST;
MLIST mylist, *listp;          /* struct mlist mylist, *listp */
```

`MLIST` ist jetzt ein Synonym für den Typ `struct mlist`, und daher auch genauso zu gebrauchen. `mylist` ist somit eine Variable vom Typ `struct mlist`, und `listp` ein Zeiger auf diesen Typ. Innerhalb der Stuktur kann man dieses Synonym jedoch noch nicht verwenden, weil die Definition der Struktur ja erst mit der schließenden geschweiften Klammer abgeschlossen ist. Die Großschreibung der *typedef*-Typen ist wieder Konvention bei Anwendungsprogrammen zum Zwecke besserer Lesbarkeit, im Bereich der Systemprogrammierung hingegen finden wir aus dem gleichen Grunde auch viele kleingeschriebene *typedef*s. Z.B. sind die Systemdatentypen `size_t`, `ptrdiff_t`, `FILE` und viele, viele andere alles *typedef*s, die dann, wie man sieht, durch besondere Namensbildung dem Programmierer kenntlich gemacht werden.

Hier noch ein berühmtes Beispiel aus der Systemprogrammierung, das zeigt, wie hilfreich *typedef* zur Verbesserung der Lesbarkeit angewandt werden kann:

```
void (*signal(int signo, void (*func)(int)))(int);
```

ist der Prototyp der Signalfunktion, so wie sie auch in ISO-C definiert[26] ist. In Worten: `signal` ist eine Funktion, die als Parameter einen int nimmt, `signo`, und einen Zeiger, `func`, auf eine Funktion, die einen int nimmt und nichts zurückgibt – das waren die beiden Parameter – und die einen Zeiger auf eine Funktion zurückgibt, die einen int nimmt und nichts zurückgibt.

Diese komplizierte Deklaration läßt sich mittels *typedef* leicht auflösen:

```
typedef void Sigfunc(int);
Sigfunc *signal(int, Sigfunc*);
```

Daraus folgt: `Sigfunc` ist der Typ einer Funktion, die einen int nimmt und nichts zurückgibt. `signal` ist eine Funktion, die einen int und einen `Sigfunc`-Zeiger nimmt, und einen ebensolchen, nämlich einen `Sigfunc`-Zeiger zurückgibt. Das wars!

[26] In signal.h. Wer das übrigens ohne weiteres lesen kann, beherrscht die C-Deklarationssyntax. Zur Erleichterung: Es gibt nicht allzuviele Funktionen von dieser Sorte!

Zum Schluß noch einige Anmerkungen zu den Beispielprogrammen:

Alle Beispielprogramme sind lauffähig. Sie wurden mittels eines Hilfsprogramms in LaTeX überführt und in den Quelltext dieser Einführung eingebunden. Sie dienen der Demonstration verschiedener syntaktischer Elemente der Sprache C und sollen ein Gefühl für den Stil vermitteln. Die Programme sind in der ISO-C90-Variante *Clean C* geschrieben, d.h. es sind gleichzeitig auch gültige C++-Programme. Die Zeilennummern dienen der Referenzierung und gehören nicht zum Quelltext.

C ist eine sog. formatfreie Sprache, im Prinzip könnte man also den ganzen Quelltext – bis auf die Präprozessoranweisungen – auf eine Zeile schreiben (evtl. begrenzt der Compiler die Zeilenlänge, sog. *environmental limits*, in C90 min. 509 Zeichen pro logischer Quelltextzeile). Das sollte man allerdings aus Lesbarkeitsgründen gar nicht erst versuchen; denn bekanntlich sind Lesen und Verstehen von Programmen ja um etliches schwerer als das Schreiben derselben. Empfohlen: Zeilenlänge ≤ 78.

Ein einheitlicher Stil, man spricht in diesem Zusammenhang auch von *Coding Conventions*, erleichtert das Verständnis und den Austausch von Quelltexten zwischen Programmierern. Äußerst wichtig sind hierbei Konsistenz in Einrückung und Formatierung. Der hier gewählte Stil, K&R, ist einer unter etlichen heute verbreiteten (1TBS, Allman/BSD, KNF, GNU). Einrückungen erfolgen mit TAB 4, die Texte enthalten aber keine TABs, damit sie auf allen Medien (Drucker, Editor, Konsole) gleich aussehen. Viele Editoren lassen sich so einstellen, daß statt der TABs Leerzeichen in den Quelltext eingesetzt werden. Weitere Kennzeichen dieses Stils – und der meisten anderen auch – sind folgende: *Ein* Leerzeichen zwischen Schlüsselwörtern und den zugehörigen Klammern, *kein* Leerzeichen zwischen Funktionsnamen und den zugehörigen Klammern der Parameterliste, eine Leerzeile zwischen den Deklarationen im Kopf eines Blocks und dem Beginn der Anweisungen. Weitere Konventionen werden oft in Firmen oder innerhalb von Projekten bis in Einzelheiten vorgeschrieben. Man gewöhne sich einen Stil an und bleibe dabei. Hat man Fremdcode zu warten, ist es empfehlenswert, sich zur Wahrung der Einheitlichkeit nach dessen Stil zu richten und nicht etwa nach eigenem Gusto umzuformatieren.

Einige Bemerkungen zur Kommentierung: Zeilenkommentare über, kurze Restzeilenkommentare auf der Anweisungszeile. Blockkommentare gebündelt vor größeren Codeabschnitten. Will man bei der Entwicklung größere Codeabschnitte auskommentieren (Schachtelungsverbot!), empfiehlt es sich, dafür den Präprozessor zu nutzen und den jeweiligen Abschnitt in `#if 0` und `#endif` einzuschließen.

Es sollte immer nur kommentiert werden, was *nicht* offensichtlich ist. Daß z.B. in einer Anweisung etwas addiert wird, sollte auch so klar sein, aber vielleicht ist es manchal hilfreich zu wissen, was, weshalb oder warum gerade hier addiert wird. Überflüssige Kommentare sind nicht nur lästig, sie sind *schädlich*. Weniger ist oft mehr! Durch geschickte Wahl der Namen von Variablen etc. kann man sich viel quälende Kommentierung sparen und damit die Lesbarkeit des Quelltextes wesentlich erhöhen! Wie bei Einführungsmaterial üblich, sind einige der Beispiele stark überkommentiert. Das ist kein Produktionsstil und sollte *nicht* nachgeahmt werden!

A Anhang

A.1 Formatierte Ausgabe mit printf()

Die Formatwandlungen der `printf()`-Familie stellen eine eigene kleine Sprache (*little language*) dar. Eine Formatwandlungsangabe (*format specifier*) beginnt mit dem Zeichen `%` und endet mit einem Umwandlungszeichen wie in folgender Tabelle:

Zeichen	Argument – Umwandlung in Darstellung
d, i	int – dezimal mit Vorzeichen
o	int – oktal ohne Vorzeichen
x, X	int – hexadezimal ohne Vorzeichen (mit a-f oder A-F)
u	int – dezimal ohne Vorzeichen
c	int – Zeichen als unsigned char
s	char* – Zeichenkette als C-String
f	double – dezimal mit Vorzeichen (0 ohne Dezimalpunkt)
e, E	double – dezimal mit Exponentialkennung
g, G	double – nach Bedarf wie f oder e, E
p	void* – als Zeiger (implementationsabhängig, meist in Hex)
%	keine Umwandlung: `%%` bewirkt Ausgabe von `%`
n	int* – Anzahl bisher ausgegeb. Zeichen wird ins Argument geschrieben

Zwischen `%` und dem Umwandlungszeichen können, falls gewünscht und sinnvoll, in Reihenfolge oder auch kombiniert, optional weitere Angaben stehen:

Angabe	Bedeutung
-	Linksausrichtung im Ausgabefeld
+	Zahlenausgabe immer mit Vorzeichen
Leerz.	falls erstes Zeichen kein Vorzeichen, Voranstellung eines Leerzeichens
0	Feldbreite mit führenden Nullen auffüllen
#	alternative Ausgabe für o, x, X, e, E, f, g und G
Zahl	Zahl gibt (minimale) Breite des Ausgabefeldes an
.	trennt Feldbreite von Genauigkeit
Zahl	Genauigkeit
h	Argument ist short oder unsigned short
l	Argument ist long oder unsigned long
L	Argument ist long double

Die (minimale) Feldbreite, wird, wenn nicht anders angegeben, je nach Ausrichtung, mit Leerzeichen aufgefüllt. Die Genauigkeit bedeutet bei Zeichenketten die maximale Anzahl der auszugebenden Zeichen, bei Ganzzahlen die minimale Anzahl von auszugebenden Ziffern (aufgefüllt wird mit führenden Nullen), Bei f, e oder E Wandlungen bedeutet die Genauigkeit die Anzahl der Nachkommastellen, bei g oder G Wandlungen die Anzahl der signifikanten Stellen. Anstelle der Zahl für Feldbreite oder Genauigkeit kann man auch einen * angeben, die entsprechende Angabe wird dann dem jeweils nächsten Argument entnommen, das vom Typ int sein muß.

A.2 Auswahl von Makros und Funktionen der Std.-Bibliothek

stddef.h:

NULL *Nullpointerkonstante: 0, 0L oder (void*)0*
size_t *vorzeichenloser Typ als Resultat des* `sizeof`*-Operators*
ptrdiff_t *vorzeichenbehafteter Typ für Pointerdifferenz*
offsetof(structtype, member) *liefert Ablage in Bytes vom Beginn der Struktur*

stdio.h:

BUFSIZ *Standardgröße des Streambuffers*
EOF *Dateiendekennung (meist -1)*
FILE *interne Struktur für Dateiverarbeitung*
FILENAME_MAX *maximale Dimension eines char-Arrays für Dateinamen*
stdin *Standardeingabekanal*
stdout *Standardausgabekanal*
stderr *Standardfehlerkanal*

```
FILE *fopen(const char *filename, const char *mode);
FILE *freopen(const char *filename, const char *mode, FILE *stream);
int fclose(FILE *stream);
int feof(FILE *stream);
int ferror(FILE *stream);
int fflush(FILE *stream);
size_t fread(void *buf, size_t size, size_t nelem, FILE *stream);
size_t fwrite(const void *buf, size_t size, size_t nelem, FILE *stream);
int fseek(FILE *stream, long offset, int mode);
```

SEEK_SET *Dateianfang*
SEEK_CUR *aktuelle Position in der Datei*
SEEK_END *Dateiende*

```
long ftell(FILE *stream);
int fgetpos(FILE *stream, fpos_t *pos);
int fsetpos(FILE *stream, const fpos_t *pos);
void rewind(FILE *stream);
int getchar(void); int getc(FILE *stream); int fgetc(FILE *stream);
int putchar(int c); int putc(int c, FILE *stream); int fputc(int c, FILE *stream);
char *fgets(char *buf, int n, FILE *stream);
int fputs(const char *buf, FILE *stream);
int scanf(const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
int sscanf(const char *buf, const char *format, ...);
int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int sprintf(char *buf, const char *format, ...);
int remove(const char *filename);
int rename(const char *oldname, const char *newname);
```

stdlib.h:

```
    EXIT_SUCCESS    unter Unix (u.v.a.) int-Wert 0
    EXIT_FAILURE    unter Unix int-Wert != 0, meist 1
    RAND_MAX    maximaler Wert der Funktion rand()
    void abort(void);     void exit(int status);
    int atexit(void (*func)(void));
    int system(const char *s);
    int abs(int i);     long labs(long i);
    int atoi(const char *s);     long atol(const char *s);
    double atof(const char *s);
    long strtol(const char *s, char **endptr, int base);
    unsigned long strtoul(const char *s, char **endptr, int base);
    double strtod(const char *s, char **endptr);
    void *malloc(size_t size);     void *calloc(size_t nelem, size_t size);
    void *realloc(void *ptr, size_t size);     void free(void *ptr);
    int rand(void);     void srand(unsigned int seed);
    div_t div(int numer, int denom);     ldiv_t ldiv(long numer, long denom);
    void *bsearch(const void *key, const void *base, size_t nelem, size_t size,
                  int (*cmp)(const void *ck, const void *ce));
    void qsort(void *base, size_t nelem, size_t size,
               int (*cmp)(const void *e1, const void *e2));
```

string.h:

```
    void *memset(void *s, int c, size_t n);
    void *memchr(const void *s,int c, size_t n);
    int memcmp(const void *s1, const void *s2, size_t n);
    void memcpy(void *s1, const void *s2, size_t n);
    void memmove(void *s1, const void *s2, size_t n);
    size_t strlen(const char *s);
    int strcmp(const char *s1, const char *s2);
    char *strchr(const char *s, int c);
    char *strstr(const char *s1, const char *s2);
    char *strcat(char *s1, const char *s2);
    char *strcpy(char *s1, const char *s2);
    char *strtok(char *s1, const char *s2);
```

time.h:

```
    CLOCKS_PER_SEC Ticks der internen Systemuhr: clock()
    clock_t clock(void);
    time_t time(time_t *tp);
    char *ctime(const time_t *tp);
    time_t maketime(struct tm *tp);
    char *asctime(const struct tm *tp);
    struct tm *gmtime(const time_t *tp);
    struct tm *localtime(const time_t *tp);
```

ctype.h:

int iscntrl(int c); int isspace(int c); int isprint(int c);
int isgraph(int c); int ispunct(int c); int isalnum(int c);
int isxdigit(int c); int isdigit(int c);
int isalpha(int c); int isupper(int c); int islower(int c);
int tolower(int c); int toupper(int c);

errno.h:

errno *globaler Makro für Speicherung von Fehlernummern*
EDOM *Domainfehler: Funktion war für Argument nicht definiert*
ERANGE *Bereichsfehler: Funktionswert außerhalb des darstellbaren Bereichs*

math.h:

HUGE_VAL *wahrsch. unendlich ...*
double sin(double x); double asin(double x);
double cos(double x); double acos(double x);
double tan(double x); double atan(double x); double atan2(double y, double x);
double sinh(double x); double cosh(double x); double tanh(double x);
double sqrt(double x); double pow(double x, double y);
double exp(double x); double log(double x); double log10(double x);
double ldexp((double x, int exp); double frexp(double x, int *pexp);
double fmod(double x, double y); double modf(double x, double *pi);
double ceil(double x); double floor(double x);

limits.h:

CHAR_BIT *Anzahl der Bit des Datentyps char*
CHAR_MAX CHAR_MIN UCHAR_MAX
SCHAR_MAX SCHAR_MIN
SHRT_MAX SHRT_MIN USHRT_MAX
INT_MAX INT_MIN UINT_MAX
LONG_MAX LONG_MIN ULONG_MAX

float.h:

FLT_RADIX FLT_ROUNDS *gilt zentral für alle FP-Operationen*
FLT_DIG FLT_MANT_DIG FLT_EPSILON
FLT_MAX FLT_MIN FLT_MAX_EXP FLT_MIN_EXP
DBL_DIG DBL_MANT_DIG DBL_EPSILON
DBL_MAX DBL_MIN DBL_MAX_EXP DBL_MIN_EXP
LDBL_DIG LDBL_MANT_DIG LDBL_EPSILON
LDBL_MAX LDBL_MIN LDBL_MAX_EXP LDBL_MIN_EXP
FLT_MAX_10_EXP FLT_MIN_10_EXP
DBL_MAX_10_EXP DBL_MIN_10_EXP
LDBL_MAX_10_EXP LDBL_MIN_10_EXP

signal.h:

void (*signal(int signo, void (*func)(int)))(int);
SIG_IGN func-*Argument: Signal* signo *soll ignoriert werden*
SIG_DFL func-*Argument: Signal* signo *defaultmäßig handhaben*

A.3 Speicherauslegung eines Unix-Prozesses

Als Prozeß wird unter Unix die Instanz eines zur Ausführung in den Speicher geladenen Programms bezeichnet. Der Prozeß läuft in einem separaten virtuellen Adreßraum, sein von ihm zur Laufzeit belegter Speicher wird in verschiedene Bereiche – auch Segmente oder Sektionen genannt – aufgeteilt:

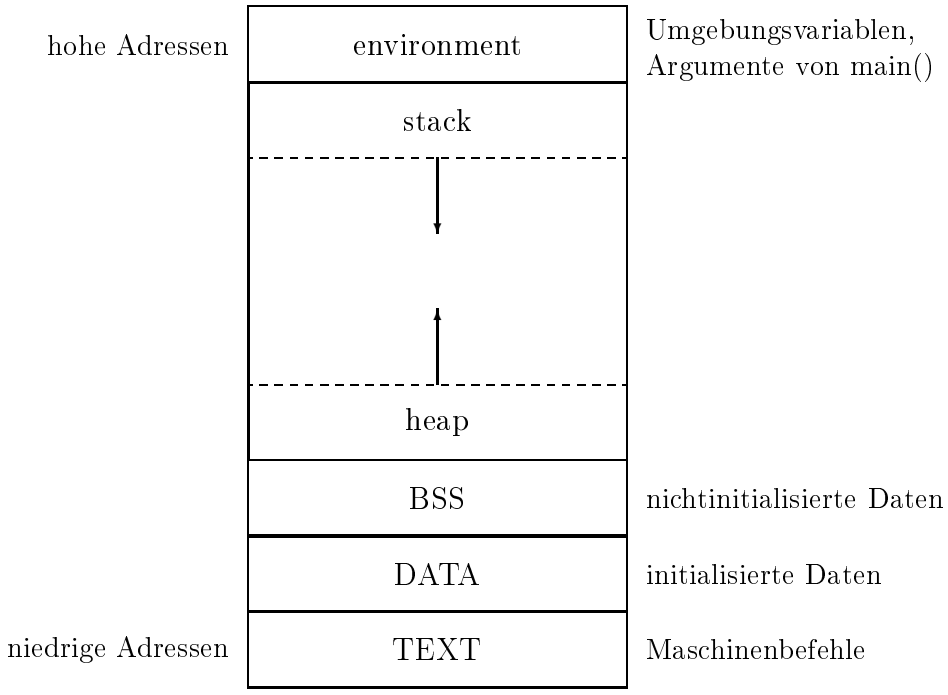

Abbildung 1: Typische Speicherauslegung eines Unix-Prozesses

- TEXT – der ausführbare Programm-Code – Maschinenbefehle, compiliert, assembliert, handcodiert, hinzugebunden, wie in der Programmdatei abgelegt.
- DATA – initialisierte Daten zum Lesen und Schreiben, wie beim TEXT, nur eben keine Maschinenbefehle, sondern Daten. Auf vielen Systemen gibt es hier noch ein besonderes Segment RODATA[27] – als Bereich für initialisierte Daten nur zum Lesen, z.B. Platz für Strings, Konstanten, etc. Auch diese Daten müssen in der Programmdatei vorhanden sein.
- BSS – Block Storage Segment – Bereich für nicht initialisierte Daten. Dieser Bereich wird nicht in der Programmdatei gespeichert, nur die Größe des Bedarfs. Unter C und Unix wird dieses Segment zum Programmstart ausgenullt bereitgestellt.

[27] Read Only DATA

- *Heap* – dieser Bereich zur Datenspeicherung wird nach Bedarf zur Laufzeit vom Programm angefordert und verwaltet. Dazu dienen die C-Bibliotheks-Funktionen `malloc()`, `calloc()`, `realloc()`, `free()` und die Systemaufrufe `brk()` und `sbrk()`. Der *Heap* wächst von unten nach oben, d.h. von niederen in Richtung zu höheren Adressen.

- *Stack* – hier werden u.a. lokale Variablen (entsprechend der Speicherklasse *auto* in C) gespeichert, Register gesichert etc. Dieser Bereich wird durch den *Stack Pointer* (SP) verwaltet, dessen Wert vom System beim Programmstart vorgegeben wird. Ein anderer Zeiger in diesen Bereich ist der *Frame Pointer* (FP), von dem aus die im *Stack Frame* liegenden Variablen angesprochen werden. Jede Routine hat ihren eigenen *Stack Frame*, der bei ihrem Aufruf aufgebaut und bei der Rückkehr wieder abgebaut wird. Dadurch ist z.B. Rekursion möglich. Der *Stack* wächst von oben nach unten, Speicherplatz dafür wird vom System nach Bedarf zur Verfügung gestellt. Die C-Funktion `alloca()`[28] reserviert z.B. Speicherplatz auf dem Stack.

- *Environment* – Der sog. Umgebungsbereich wird dem Prozeß vom Betriebssystem zur Verfügung gestellt. Hier liegen die Strings der Umgebungsvariablen. Auch die Argumente, die dem Programm auf der Kommandozeile mitgegeben werden, befinden sich in diesem Bereich. Diese sind üblicherweise unter `*argv[]` erreichbar, das *Environment* läßt sich mit der Funktion `getenv()` aus `stdlib.h` auslesen, mit der man Dinge wie z.B. `PATH=`, `HOME=` etc. erreichen kann.

  Will man auch schreibend auf diese Umgebung zugreifen, muß man sich der POSIX[29]-konformen Systemvariablen- und -funktionen bedienen, deren Schnittstelle unter Unix in der System-Header-Datei `unistd.h` zur Verfügung gestellt wird, z.B. `*environ[]`, `setenv()`.

[28] Nicht in ISO-C, aber in vielen Erweiterungen enthalten.
[29] Portable Operating System Interface (for uniX-like systems ...)

A.4 Ganzzahloperationen auf Maschinenebene

Es gehört zwar nicht direkt zum Thema C, aber für Benutzer einer Programmiersprache mit ihren vielfältigen Operatoren auf Bitebene ist es sicher vorteilhaft, sich mit der Abbildung der verschiedenen Datentypen im Speicher und den möglichen Operationen auf Maschinenebene genauer zu befassen. Da heutzutage die weitaus meisten Prozessoren ihre Speicher Byte-adressiert – mit 8-Bit Bytes – ansprechen und sich die interne Ganzzahldarstellung im Zweierkomplement aus technischen Gründen allgemein durchgesetzt hat, wird hier nur auf diese eingegangen.

Für die Ablage von Multibyte-Datentypen im Speicher gibt es zwei Hauptarchitekturen: *big endian*[30] und *little endian*[31]. Manche Prozessoren[32] beherrschen beide Methoden und lassen sich umschalten, manchmal sogar im laufenden Betrieb, aber meist legt das jeweilige Betriebssystem eine dieser beiden Techniken vom Start an fest. Beide Methoden sind technisch durchaus gleichwertig, man muß sich nur darauf einigen, wie man es halten will. Die Netzbytefolge[33] (*net byte order*) bei TCP/IP ist z.B. standardmäßig auf Big Endian festgelegt, damit alle Maschinen miteinander kommunizieren können. Diejenigen, die intern die andere Methoden verwenden, müssen dann eben umwandeln.

Multibyte-Daten werden im Speicher grundsätzlich von niederen in Richtung höherer Adressen ausgelegt. Was bedeuten nun diese Begriffe[34]? *Big Endian* speichert Multibyte-Datentypen, in C z.B. `short`, `int`, `long`, `double`, Pointer etc. mit dem dicken Ende, also dem höchstwertigen Byte zuerst, *Little Endian* macht es umgekehrt. Zur Illustration betrachte man die beiden Abbildungen.

adr-1	adr	adr+1	adr+2	adr+3	adr+4
	0x12	0x34	0x56	0x78	

Abbildung 2: 0x12345678 als 32-Bit-Ganzzahl im *Big Endian*-Speicherformat

adr-1	adr	adr+1	adr+2	adr+3	adr+4
	0x78	0x56	0x34	0x12	

Abbildung 3: 0x12345678 als 32-Bit-Ganzzahl im *Little Endian*-Speicherformat

[30] z.B. IBM System /360 ff., PowerPC, SPARC, Motorola 68K und 88K

[31] z.B. DEC VAX und Intel x86

[32] z.B. MIPS und Intel i860

[33] siehe dazu auch den berühmten Artikel von Danny Cohen, On Holy Wars and a Plea for Peace, IEEE Computer, Oct 1981, pp.48-54

[34] Sie beziehen sich auf den satirischen Roman von Jonathan Swift, Gullivers Reisen, in dem vom Krieg der Liliputaner mit den Leuten von Blefuscu, den Spitzendern und den Breitendern die Rede ist, nämlich denjenigen, die das Ei am spitzen Ende aufschlagen mit denjenigen, die dies am breiten Ende tun.

Wie auch immer die Repräsentation solcher Multibyte-Werte im Speicher aussieht, zur Verarbeitung müssen sie in Register der CPU geladen werden, und dort sieht die Bit-Auslegung dann wieder regelmäßig so aus, bzw. wird so dargestellt, wie es unserer traditionellen Schreibweise eines dualen Stellenwertsystems entspricht, mit der Bit-Numerierung von 0 bis 31 von rechts nach links, gemäß der Wertigkeit der Bits. Hier als Beispiel das Abbild eines 32-Bit-Registers:

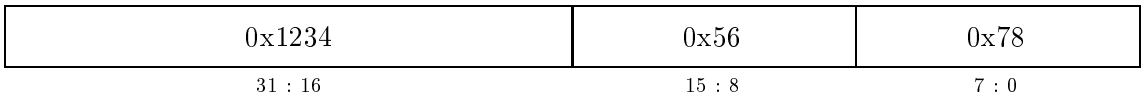

Abbildung 4: Ein 32-Bit-Register mit dem Inhalt 0x12345678

C kennt keine Binärnotation von Konstanten, aus der Oktal- oder Hexdarstellung läßt sich jedoch das Binärformat leicht erzeugen, wenn man sich merkt, daß eine Oktalziffer immer drei Bit bündelt und eine Hexziffer vier. Die Bitdarstellung aus der Dezimalnotation zu generieren, ist etwas schwieriger, weil 10 keine Zweierpotenz ist. Bei größeren Werten sind Oktal- und, besser noch, Hexdarstellung übersichtlicher, weil man beim Lesen (oder Schreiben) von langen Ketten von Nullen und Einsen leicht Fehler machen kann. Einige Beispiele mögen dies verdeutlichen:

Dezimal	Bin 8	Bin 16	Okt 8	Okt 16	Hex 8	Hex 16
1	00000001	0000000000000001	001	000001	01	0001
10	00001010	0000000000001010	012	000012	0A	000A
64	01000000	0000000001000000	100	000100	40	0040
100	01100100	0000000001100100	144	000144	64	0064
127	01111111	0000000001111111	177	000177	7F	007F
128	10000000	0000000010000000	200	000200	80	0080
255	11111111	0000000011111111	377	000377	FF	00FF
1000	--	0000001111101000	--	001750	--	03E8
10000	--	0010011100010000	--	023420	--	2710
32767	--	0111111111111111	--	077777	--	7FFF
65535	--	1111111111111111	--	177777	--	FFFF

Anhand dieser Beispiele einer Interpretation der Bit als Stellen in einem dualen Stellenwertsystem mit Begrenzung der Anzahl der Stellen, läßt sich auch leicht ersehen, daß man bei einer arithmetischen Operation, wie z.B. der Addition von 1 auf den höchstmöglichen Wert wieder bei 0 ankommt, bzw. bei Subtraktion von 0 den im jeweiligen Format höchstmöglichen Wert erreicht. Diese Erscheinung nennt man auch *wrap around*. Es handelt sich hier also um Modulus-Arithmetik, d.h. für das jeweilige Format gilt, wenn n die Anzahl der Stellen ist, so beträgt der höchstmögliche Wert $2^n - 1$ und es werden arithmetische Operationen modulo 2^n ausgeführt. Dieses Verhalten ist so im Standard der Sprache C vorgesehen und vorgeschrieben.

Um auch negative Ganzzahlen darstellen zu können, gibt es verschiedene Systeme, z.B. Vorzeichen und Betrag, Einerkomplement und Zweierkomplement. Von

diesen hat sich aus technischen Gründen das letztere durchgesetzt, sodaß wir unsere Erörterung darauf beschränken werden.

Bei der Zahlendarstellung im Zweierkomplement wird das Bit an der höchsten Stelle als Vorzeichen betrachtet. Ist es 0, gilt die Zahl als positiv und die übrigen Stellen werden in bekannter Weise interpretiert, ist es 1, gilt sie als negativ, und der Wert ergibt sich, indem man alle Stellen komplementiert, zu diesem dann 1 addiert, und das Ergebnis als Betrag der negativen Zahl interpretiert. Hier einige Beispiele für die Darstellung negativer Ganzzahlen im Zweierkomplement:

Dezimal	Bin 8	Bin 16	Okt 8	Okt 16	Hex 8	Hex 16
-1	11111111	1111111111111111	377	177777	FF	FFFF
-2	11111110	1111111111111110	376	177776	FE	FFFE
-3	11111101	1111111111111101	375	177775	FD	FFFD
-4	11111100	1111111111111100	374	177774	FC	FFFC
-10	11110110	1111111111110110	366	177766	F6	FFF6
-100	10011100	1111111110011100	634	177634	9C	FF9C
-127	10000001	1111111110000001	601	177601	81	FF81
-128	10000000	1111111110000000	200	177200	80	FF80
-1000	--	1111110000011000	--	176030	--	FC18
-10000	--	1101100011110000	--	154360	--	D8F0
-32767	--	1000000000000001	--	400001	--	8001
-32768	--	1000000000000000	--	400000	--	8000

Bei Speicherung der Ganzzahlen im Zweierkomplement gilt: Wenn n die Anzahl der Bit ist, so erstreckt sich der darstellbare Bereich von -2^{n-1} bis $2^{n-1}-1$, wobei die scheinbare Asymmetrie dadurch zustandekommt, daß die Null (Vorzeichenbit ist 0, alle anderen Bit ebenfalls) zur positiven Seite gezählt wird.

Bei vorzeichenbehafteten Ganzzahldatentypen in C wird das Verhalten beim Überschreiten des möglichen in der entsprechenden Größe darstellbaren Wertes vom Standard als implementationsabhängig bezeichnet. Bei der hier beschriebenen Zweierkomplementdarstellung kommt man bei Überlauf des höchstmöglichen positiven zum betragsmäßig größten negativen Wert.

Versucht man einen Wert zu speichern, der das Fassungsvermögen einer vorgegebenen Speichergröße übersteigt, so wird der höherwertige Teil meist einfach abgeschnitten. Was bei Bitoperationen passiert, kann man mit diesem Modell ebenfalls gut visualisieren. Das Verhalten bei Operationen auf negativen Werten wird vom Standard in vielen Fällen als implementationsabhängig bezeichnet.

Ob Überlauf und Unterlauf im Wertebereich der beteiligten Speichergrößen auftreten können und was dann zu tun ist, wird im Ganzzahlbereich auf Maschinenebene meist nicht besonders behandelt und liegt (in C, wie in Assembler und vielen weiteren Programmiersprachen) ganz allein in der Verantwortung des Programmierers. Einzig die Division durch Null bewirkt, von der Hardware ausgelöst, meist ein das Programm terminierendes Signal, das aber von diesem abgefangen werden kann.

A.5 Gleitpunktoperationen auf Maschinenebene

Gleitpunktzahlen im Computer sollen ein Abbild der reellen Zahlen in der Mathematik sein, dieses Abbild weist aber aufgrund des eingeschränkten Speicherplatzes erhebliche Mängel auf. In 32 Bit, beispielsweise, lassen sich eben nur 2^{32}, also gut vier Milliarden, verschiedene Werte darstellen. Bei den Ganzzahlen sind die darstellbaren Werte gleichmäßig über den ganzen Bereich verteilt, und die Grenzen der Darstellung sind gut absehbar. Bei dem riesigen Wertebereich der Gleitpunktzahlen, z.B. von etwa $1.175 \times 10^{-38} \ldots 3.403 \times 10^{38}$, dazu noch einmal das gleiche in negativ, bei gleichem Speicherverbrauch von 32 Bit muß es also entspr. Lücken geben. Dazu ist die Verteilung ungleichförmig, relativ dicht in der Nähe der Null mit riesigen Lücken am äußeren Ende. Man erkauft also die enorme Erweiterung des Wertebereichs mit einem Verlust an Genauigkeit.

Gleitpunktoperationen auf Systemen mit beschränkter Genauigkeit sind äußerst komplex. Das Verhalten bei Rundung, Überlauf und Unterlauf im darstellbaren Wertebereich, Genauigkeitsverlust, unerlaubten Operationen etc. sorgt oft für manch böse Überraschung beim gutgläubigen Programmierer (oder Anwender).

Die Gleitpunktformate nach IEEE 754

In der Anfangszeit der Computer hatte fast jeder Hersteller von Hard- und Software (Compiler) sein eigenes Format, sodaß es äußerst schwierig war und oft auch zu inkonsistenten Ergebnissen führte, Programme von einer Maschine auf eine andere zu übertragen. Ende der 1970er Jahre begann dann auf Inititive der ACM[35] und dann der IEEE[36] der Anlauf zu einer Standardisierung[37] der Formate und des numerischen Verhaltens der binären Gleitpunktarithmetik für Microprozessoren, der die im Laufe der vorangegangenen Jahrzehnte gemachten Erfahrungen zusammenfaßte und im Hinblick auf die Anforderungen der meist wissenschaftlichen Nutzer auch weiterentwickelte. Hier kurz die historische Entwicklung:

1981	Entwurf des Std. zur binären Gleitpunktarithmetik	Draft IEEE 754
1985	Annahme des Std. zur binären Gleitpunktarithmetik	IEEE 754
1987	Fassung als basisunabhängige Gleitpunktarithmetik	IEEE 854
1989	Übernahme des Std. IEEE 754 durch die IEC	IEC 60559:1989

IEEE 754 spezifiziert und definiert 4 Formate:

einfache Genauigkeit	*single precision*	32 Bit
doppelte Genauigkeit	*double precision*	64 Bit
einfache erweiterte Genauigkeit	*single extended precision*	$\geq$ 43 Bit
doppelte erweiterte Genauigkeit	*double extended precision*	$\geq$ 79 Bit

[35] Association for Computing Machinery

[36] Institute of Electrical and Electronics Engineers

[37] Der genaue Name des Standards ist: IEEE Standard for Binary Floating-Point Arithmetic for microprocessor systems (ANSI/IEEE Std 754-1985), danach auch bekannt als IEC 60559:1989.

Die ersten beiden Formate sind bis auf die Bitebene genau spezifiziert, bei den erweiterten Formaten ist nur eine Mindestbitzahl vorgegeben, und die Aufteilung etc. bleibt dem Implementor überlassen. Um dem Standard zu genügen, reicht es aus, das erste Format bereitzustellen. Das einfache erweiterte Format wird kaum je verwendet, das doppelte erweiterte hingegen ist in vielfältigen Ausführungen als internes Format von mathematischen Coprozessoren (von 80 bis 128 Bit) in Hardware implementiert worden.

Die Gleitpunktdatentypen vieler Programmiersprachen lassen sich leicht auf den IEEE-Typen abbilden. So entspricht *single precision* dem FORTRAN `REAL` und dem C `float`, *double precision* entspr. dem FORTRAN `DOUBLE PRECISION` und dem C-Typ `double`.

Weite Verbreitung erreichte der Standard vor allem, weil Hesteller wie Intel[38] und Motorola[39] Coprozessoren herausbrachten, die mehr oder weniger gut den IEEE 754 oder zumindest wichtige Teile davon implementierten. Diese Coprozessoren boten die beiden Hauptformate an und arbeiteten intern zur Erhöhung der Genauigkeit mit dem erweiterten 80-Bit-Format.

Hier ein kleiner Überblick über die Auslegung der Formate:

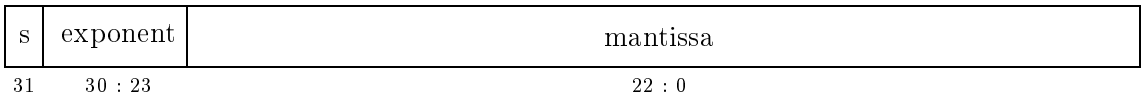

Abbildung 5: Das Format IEEE Single Precision

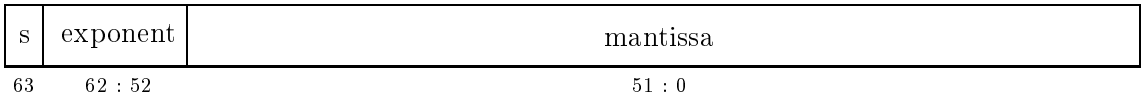

Abbildung 6: Das Format IEEE Double Precision

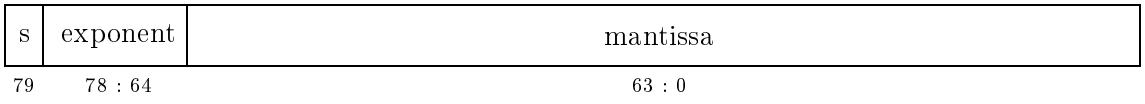

Abbildung 7: IEEE Double Extended Precision (Intel/Motorola-Version 80-Bit)

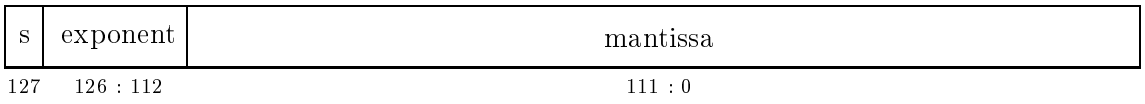

Abbildung 8: IEEE Double Extended Precision (SPARC-Version 128-Bit)

Die Mantisse, auch *fraction* oder offiziell *significand* genannt, wird als links normalisierte Binärbruchzahl abgelegt, sodaß gilt $1 \geq m < 2$, und da bei der Linksnormalisierung die 1 vor dem gedachten Binärpunkt immer vorhanden ist, braucht

[38] Intel brachte 1980, also schon zum Ende des Draft 754, den Coprozessor 8087 heraus.

[39] Motorola folgte darauf mit dem 68881, der fast den kompletten IEEE 754 Std. implementierte.

man sie nicht zu speichern und kann somit noch ein Bit an Genauigkeit gewinnen. Bei den *extended*-Formaten verzichtet man meist auf dies gedachte Bit. Zur Abmilderung der Rundungsfehler gibt es auch bei den 32 und 64-Bit-Formaten meist noch drei sog. versteckte Bit (*hidden bits*) noch unter der niederwertigsten Stelle der sichtbaren Mantisse, sog. *guard bits: guard bit, round bit, sticky bit.*

Der Exponent wird, hauptsächlich um Vergleiche zu beschleunigen, als vorzeichenlose Binärzahl mit Additionsverschiebung ins Positive (*bias*) gespeichert, sie beträgt $2^{e-1}-1$, wobei e die Anzahl der Exponent-Bit ist, also z.B. 127 bei einem 8-Bit Exponenten im IEEE *single precision* Format. Die in den Abbildungen der Einfachheit halber *exponent* genannte Position, wird daher auch oft *biased exponent* genannt und heißt offiziell Charakteristik (*characteristic*).

Das Vorzeichen wird in dem Bit s gespeichert. Ist es 0, so ist die Zahl positiv, andernfalls negativ. Dies ist also eine Codierung als Vorzeichen und Betrag.

Für Sonderfälle gibt es noch folgende Spezialcodierungen:

exp bits	mantissa bits	Bedeutung
alle 0	alle 0	Gleitpunkt Null (exakt 0.0)
alle 0	nicht Null	denormalisiert (gradueller Unterlauf)
alle 1	alle 0	unendlich
alle 1	nicht Null	NaN (Not a Number), d.h. ungültiger Wert

Auf diese Weise gibt es also zwei verschiedene Nullen, $+0$ und -0, die aber beim Vergleich als gleich gelten. Der Vergleich auf Gleichheit ist beim Gleitpunktformat äußerst heikel: Was in der Mathematik gilt, sieht bei der Unvollkommenheit der Darstellung im Computer[40] oft ganz anders aus!

IEEE schreibt auch das Vorhandensein verschiedener Rundungsmodi vor:

0 zum nächsten, bei gleichem Abstand zum geraden Wert
1 nach Null, d.h. abschneiden (*truncate*)
2 nach $+\infty$ aufrunden
3 nach $-\infty$ abrunden

Der Rundungsmodus 0 soll defaultmäßig eingeschaltet sein. Durch Anwenden verschiedener Rundungsmodi bei komplexen Rechnungen kann man manchmal herausfinden, wie empfindlich oder stabil sie gegenüber Rundungsfehlern sind. Für viele Hochsprachen, und natürlich gerade auch für C, werden vom System Routinen zur Verfügung gestellt, die das Umschalten der Rundungsmodi u.ä. erlauben.

Im Bereich der Gleitpunktzahlen nach IEEE können bei Fehlern in der numerischen Verarbeitung vielfältige Signale ausgelöst werden, die aber meist abgeschaltet sind. Die Division durch Null hingegen ist rechnerseitig meist erlaubt und ergibt dann unendlich. Eine weitergehende Erörterung verbietet sich aus Platzgründen, man bediene sich der reichhaltig vohandenen Literatur.

[40] Sehr zu empfehlen ist hier der renommierte Artikel von David Goldberg: What Every Computer Scientist Should Know About Floating Point Arithmetic, (bei Sun auf dem Doc-Server zu finden).

A.6 Beispielprogramme

Programm 1

```
 1: /* ranges.c    Datentypgroessen und Wertebereiche    br 8/92 */
 2: /* ------------------------------------------------------- */
 3:
 4: #include <stdio.h>
 5: #include <limits.h>
 6: #include <float.h>
 7:
 8: int main(void)
 9: {
10:     printf("\n sizes and ranges on this machine:\n");
11:
12:     printf("\nsizeof (char)  : %lu", sizeof(char));
13:     printf("\nrange          : %12d ... %12d", CHAR_MIN, CHAR_MAX);
14:     printf("\nsigned range   : %12d ... %12d", SCHAR_MIN, SCHAR_MAX);
15:     printf("\nunsigned range : %12d ... %12d\n", 0, UCHAR_MAX);
16:
17:     printf("\nsizeof (short) : %lu", sizeof(short));
18:     printf("\nsigned range   : %12d ... %12d", SHRT_MIN, SHRT_MAX);
19:     printf("\nunsigned range : %12d ... %12d\n", 0, USHRT_MAX);
20:
21:     printf("\nsizeof (int)   : %lu", sizeof(int));
22:     printf("\nsigned range   : %12d ... %12d", INT_MIN, INT_MAX);
23:     printf("\nunsigned range : %12u ... %12u\n", 0, UINT_MAX);
24:
25:     printf("\nsizeof (long)  : %lu", sizeof(long));
26:     printf("\nsigned range   : %12ld ... %12ld", LONG_MIN, LONG_MAX);
27:     printf("\nunsigned range : %12lu ... %12lu\n", 0L, ULONG_MAX);
28:
29:     printf("\nsizeof (float) : %lu", sizeof(float));
30:     printf("\nrange          : %e ... %e\n", FLT_MIN, FLT_MAX);
31:
32:     printf("\nsizeof (double): %lu", sizeof(double));
33:     printf("\nrange          : %e ... %e\n", DBL_MIN, DBL_MAX);
34:
35:     printf("\nsizeof (long double) : %lu", sizeof(long double));
36:     printf("\nrange          : %Le ... %Le\n", LDBL_MIN, LDBL_MAX);
37:
38:     return 0;
39: }
40: /* ------------------------------------------------------- */
```

Programm 2

```
 1: /* showfloat.c  show float & double formats in hex       br 8/92 */
 2: /* version for big & little endian architectures                 */
 3: /* ------------------------------------------------------------ */
 4:
 5: #include <stdio.h>
 6:
 7: int main(void)
 8: {
 9:     union {float y; int z;} x;   /* sizeof(float) == sizeof(int) */
10:     union {double y; unsigned char a[sizeof(double)];} dbits;
11:     int i, j, endian=0x12345678, *ip=&endian;
12:
13:     for (i = 0; i <= 10; i++) {
14:         x.y = i;
15:         printf("\nfloat:  %10f  hex: %08X", x.y, x.z);
16:     }
17:     putchar('\n');
18:
19:     if (*(char*)ip == 0x12)
20:         printf("\nthis machine has a big endian architecture\n");
21:     else if (*(char*)ip == 0x78)
22:         printf("\nthis machine has a little endian architecture\n");
23:     else {
24:         printf("\nthis machine has an unknown architecture\n");
25:         return 1;
26:     }
27:
28:     for (i = 0; i <= 10; i++) {                    /* char array */
29:         dbits.y = i;
30:         printf("\ndouble: %10f  hex: ", dbits.y);
31:         if (*(char*)ip == 0x12)
32:             for (j=0; j<sizeof(double); j++)       /* big endian */
33:                 printf("%02X", dbits.a[j]);
34:         else
35:             for (j=sizeof(double)-1; j>=0; j--) /* little endian */
36:                 printf("%02X", dbits.a[j]);
37:     }
38:     putchar('\n');
39:     return 0;
40: }
41: /* ------------------------------------------------------------ */
```

Programm 3

```
 1: /* xdump.c simple unix dump filter for hex & ascii  br 1/95 */
 2: /* ------------------------------------------------------- */
 3:
 4: #include <stdio.h>
 5: #include <ctype.h>
 6:
 7: #define BPL 16                  /* bytes per line of output */
 8:
 9: int main(void)
10: {
11:     int i, j, n;
12:     unsigned char buf[BPL];
13:
14:     for (i=0; n = fread(buf, 1, BPL, stdin); i+=BPL) {
15:         printf("\n%04X  ", i);              /* line number */
16:
17:         for (j=0; j<n; j++)
18:             printf(" %02X", buf[j] & 0xFF);    /* one byte */
19:
20:         for (j=0; j < BPL-n; j++)
21:             printf("   ");       /* adjust incomplete line */
22:
23:         printf("    ");             /* some filler columns */
24:
25:         for (j=0; j<n; j++)
26:             if (isprint(buf[j]))
27:                 putchar(buf[j]);         /* print it as is */
28:             else
29:                 putchar('.');            /* or print a dot */
30:     }
31:     putchar('\n');
32:
33:     return 0;
34: }
35: /* ------------------------------------------------------- */
```

Programm 4

```
 1: /* detab.c - filter replacing TABs with SPACEs    br 2/92  */
 2: /* ------------------------------------------------------- */
 3:
 4: #include <stdio.h>
 5: #include <stdlib.h>
 6:
 7: #define TAB 4                       /* default TAB setting */
 8:
 9: int main(int argc, char *argv[])
10: {
11:     int c, nspc, tab=TAB, col=1;
12:
13:     if (argc > 2) {
14:         fprintf(stderr, "usage: %s [tabwidth]\n", argv[0]);
15:         exit(1);
16:     }
17:
18:     if (argc == 2)                          /* get tabwidth */
19:         tab = abs(atoi(argv[1]));
20:
21:     if (tab<2 || tab>20)
22:         tab = TAB;                 /* force sensible value */
23:
24:     while ((c = getchar()) != EOF) {
25:         if (c == '\t')
26:             for (nspc=tab-(col-1)%tab; nspc>0; ++col,--nspc)
27:                 putchar(' ');
28:         else {
29:             putchar(c);
30:             c == '\n' ? (col=1) : ++col;
31:         }
32:     }
33:
34:     return 0;
35: }
36: /* ------------------------------------------------------- */
```

Programm 5

```
 1: /* umlaut.c - Umsetzung der deutschen Sonderzeichen      br 7/92 */
 2: /*        Ungeeignet für Texte mit Komplettgroßschreibung!       */
 3: /*   Demo versch. Methoden für Systeme mit char == signed char   */
 4: /* ------------------------------------------------------------ */
 5:
 6: #include <stdio.h>
 7:
 8: int main(void)
 9: {
10:     int c;
11:
12:     while ((c = getchar()) != EOF) {
13:         switch (c) {
14:             case 0xc4: /* 'Ä' */  /* Zeichenangabe gleich in Hex */
15:                 putchar('A'); putchar('e'); break;
16:
17:             case 0xe4: /* 'ä' */
18:                 putchar('a'); putchar('e'); break;
19:
20:             case 'Ö' & 0xff:  /* propagiertes Vorzeichen löschen */
21:                 putchar('O'); putchar('e'); break;
22:
23:             case 'ö' & 0xff:
24:                 putchar('o'); putchar('e'); break;
25:
26:             case (unsigned char) 'Ü': /* unsigned char erzwingen */
27:                 putchar('U'); putchar('e'); break;
28:
29:             case (unsigned char) 'ü':
30:                 putchar('u'); putchar('e'); break;
31:
32:             case (unsigned char) 'ß':
33:                 putchar('s'); putchar('s'); break;
34:
35:             default:
36:                 putchar(c);                 break;
37:         }
38:     }
39:     return 0;
40: }
41: /* ------------------------------------------------------------ */
```

Programm 6

```
 1: /* primfact.c - print prime factors of argument  br 4/93 */
 2: /* ------------------------------------------------- */
 3:
 4: #include <stdio.h>
 5: #include <stdlib.h>
 6: #include <math.h>
 7:
 8: int main(int argc, char *argv[])
 9: {
10:     unsigned long x, d, root;
11:
12:     if (argc != 2) {
13:         fprintf(stderr, "usage: %s ulong\n", argv[0]);
14:         return 1;
15:     }
16:     x = strtoul(argv[1], 0, 10);
17:     if (x <= 1)                              /* useless */
18:         return 1;
19:     printf("%lu: ", x);
20:     while (!(x%2)) {                           /* try 2 */
21:         x /= 2;
22:         putchar('2');
23:         putchar(' ');
24:         fflush(stdout);
25:     }
26:     root = sqrt((double)x) + 1;
27:     for (d=3; d<root; d+=2) {
28:         while (!(x%d)) {                   /* try 3 & up */
29:             x /= d;
30:             printf("%lu ", d);
31:             fflush(stdout);
32:             root = sqrt((double)x) + 1;
33:         }
34:     }
35:     if (x > 1)
36:         printf("%lu", x);
37:     putchar('\n');
38:
39:     return 0;
40: }
41: /* ------------------------------------------------- */
```

Programm 7

```
 1: /* array.c - Demo for arrays and pointers                br 7/92 */
 2: /* ------------------------------------------------------------ */
 3:
 4: #include <stdio.h>
 5:
 6: int main(void)
 7: {
 8:     int i, j, k;
 9:     int ai[24], a2i[6][4], a3i[4][3][2];
10:     int *pi, *pii, *piii, (*p2i)[4], (*p3i)[3][2];
11:
12:     pi = ai;
13:     pii = (int*) a2i;
14:     piii = (int*) a3i;
15:     p2i = a2i;
16:     p3i = a3i;
17:     pii = &a2i[0][0];
18:     piii = &a3i[0][0][0];
19:
20:     for (i=0; i<24; i++)
21:         ai[i] = pii[i] = piii[i] = i;
22:
23:     for (i=0; i<24; i++)
24:         printf(" %d", pi[i]);
25:     putchar('\n');
26:
27:     for (i=0; i<6; i++)
28:         for (j=0; j<4; j++)
29:             printf(" %d", a2i[i][j]);
30:     putchar('\n');
31:
32:     for (i=0; i<4; i++)
33:         for (j=0; j<3; j++)
34:             for (k=0; k<2; k++)
35:                 printf(" %d", a3i[i][j][k]);
36:     putchar('\n');
37:
38:     return 0;
39: }
40: /* ------------------------------------------------------------ */
```

Programm 8

```
 1: /* charfilt.c   - char filter  demo -           br 5/92 */
 2: /* copy stdin to stdout using std char I/O functions    */
 3: /* ------------------------------------------------- */
 4:
 5: #include <stdio.h>
 6:
 7: int main(void)
 8: {
 9:     int c;
10:
11:     while ((c=getchar())!= EOF) {
12:
13:                                  /* insert action here */
14:
15:         putchar(c);
16:     }
17:     return 0;
18: }
19: /* ------------------------------------------------- */
```

Programm 9

```
 1: /* linefilt.c   - line filter  demo -           br 5/92 */
 2: /* copy stdin to stdout using std line I/O functions    */
 3: /* for line organized streams only, i.e. text files     */
 4: /* ------------------------------------------------- */
 5:
 6: #include <stdio.h>
 7:
 8: int main(void)
 9: {
10:     char buf[BUFSIZ];
11:
12:     while (fgets(buf, BUFSIZ, stdin)) {
13:
14:                                  /* insert action here */
15:
16:         fputs(buf, stdout);
17:     }
18:     return 0;
19: }
20: /* ------------------------------------------------- */
```

Programm 10

```
 1: /* einmal.c  kleines Einmaleins nach stdout   br 11/89 */
 2: /* ------------------------------------------------- */
 3:
 4: #include <stdio.h>
 5:
 6: int main(void)
 7: {
 8:     int i, j;
 9:
10:     putchar('\n');
11:     for (i=1; i<=10; ++i) {
12:         for (j=1; j<=10; ++j)
13:             printf("%4d", i*j);
14:         putchar('\n');
15:     }
16:     putchar('\n');
17:     return 0;
18: }
19: /* ------------------------------------------------- */
```

Programm 11

```
 1: /* blockcpy.c   - file copying  demo -          br 5/92 */
 2: /* copy source to target using std block I/O functions  */
 3: /* return values: success: 0, error: 1, call syntax: 2  */
 4: /* ------------------------------------------------- */
 5:
 6: #include <stdio.h>
 7:
 8: int block_cpy(char *qdat, char *zdat);
 9:
10: int main(int argc, char *argv[])
11: {
12:     if (argc < 3) {
13:         fprintf(stderr,
14:                 "usage: %s  <sourcefile> <targetfile>\n",
15:                 argv[0]);
16:         return 2;
17:     }
18:     return block_cpy(argv[1], argv[2]);
19: }
20: /* ------------------------------------------------- */
```

```
21: /* ------------------------------------------------- */
22: /* block_cpy() returns 0 on success, 1 on error      */
23: /*             writes error messages to stderr       */
24: /* ------------------------------------------------- */
25:
26: int block_cpy(char *qdat, char *zdat)
27: {
28:     FILE *qfp, *zfp;
29:     int n;
30:     char buf[BUFSIZ];
31:
32:     if ((qfp=fopen(qdat, "r")) == 0) {
33:         perror(qdat);
34:         return 1;
35:     }
36:
37:     if ((zfp=fopen(zdat, "w")) == 0) {
38:         perror(zdat);
39:         fclose(qfp);
40:         return 1;
41:     }
42:
43:     while ((n=fread(buf, 1, sizeof buf, qfp)) > 0)
44:         if (fwrite(buf, 1, n, zfp) != n) {
45:             perror("fwrite");
46:             fclose(zfp);
47:             fclose(qfp);
48:             return 1;
49:         }
50:
51:     if (!feof(qfp) || ferror(qfp)) {
52:         fprintf(stderr, "read error on %s\n", qdat);
53:         fclose(zfp);
54:         fclose(qfp);
55:         return 1;
56:     }
57:
58:     fclose(zfp);
59:     fclose(qfp);
60:     return 0;
61: }
62: /* ------------------------------------------------- */
```

Literatur

[ISO90] International Organization for Standardization (ISO), ISO/IEC Standard 9899:1990, Genf, 1990
Der definitive Text des C-Standards, nur erhältlich bei der entspr. Standardorganisation. Aus anderen Quellen gibt es meist nur die jeweils letzte Entwurfsfassung (Draft). Dieser Standard wird in der Literatur oft kurz als ANSI C, ISO C, C89, C90, manchmal auch unter Einbeziehung der geringfügigen Erweiterung von 1994 als C94 oder C95 bezeichnet. Inzwischen ist die Entwicklung bei C99 (früher C9X genannt) angekommen, d.h. dem Standard ISO/IEC 9899:1999, aber zu diesem Zeitpunkt (Beginn 2007) haben die meisten Compiler – auch der gcc – noch nicht die volle Umsetzung dieses Standards erreicht.

[GNUCC] Free Software Foundation, Using and Porting GNU CC, die aktuelle, bzw. zum benutzten Compiler gehörende Version, Boston, MA,
Das Handbuch zum GCC. Enthält u.a. die genaue Beschreibung aller Optionen und eine Fülle nützlicher Hinweise.

[GLIBC] Free Software Foundation, The GNU C Library Reference Manual, die jeweils aktuelle, bzw. zur benutzten Bibliothek gehörende Version, Boston, MA,
Das Handbuch zur GNU C Library. Enthält u.a. präzise Beschreibungen aller Funktionen und Makros, Hinweise zu ihrer Benutzung, oft mit ausgezeichneten Beispielen.

[Kern89] Kernighan, Brian W., and Dennis M. Ritchie, The C Programming Language, Second Edition, ANSI C, Prentice-Hall, 1989
»Das C-Buch«. D.M.Ritchie ist der Autor der Sprache C. Das Buch beschreibt ANSI-C 89 und gibt ein Gefühl für das Flair und die Eleganz dieser heute wohl am weitesten verbreiteten Systemsprache.

[Kern90] Kernighan, Brian W., and Dennis M. Ritchie, Programmieren in C, Zweite Ausgabe, ANSI C, Hanser, 1990
Die – im Gegensatz zur ersten Ausgabe – sehr gelungene deutsche Übersetzung des obigen Standardwerks.

[Kern99] Kernighan, Brian W., and Rob Pike, The Practice of Programming, Addison-Wesley, 1999
Unter dem Motto »Simplicity – Clarity – Generality« stellen die Autoren wichtige Aspekte der Programmentwicklung in verschiedenen Programmiersprachen (C, C++, Java, AWK und Perl) garniert mit praktischen Codebeispielen unter stilistischen, modularen und Effizienzgesichtspunkten dar.

[Harb95] Harbison, Samuel P., and Guy L. Steele Jr., C, A Reference Manual, 4th edition, Prentice-Hall, 1995
Renommiertes Standardwerk zu C, enthält schon die Erweiterungen von C94.

[Ratio89] ANSI Committee X3J11, Rationale for the ANSI C Programming Language, Silicon Press, Summit, NJ, 1990
Die Rationale (Begründung) stand im Vorspann zum ANSI Standard vom Dezember 1989, obwohl selbst nicht zum Standard gehörig. Als dieser dann fast identisch (mit nur geringfügigen Änderungen in Bezeichnung und Numerierung der Abschnitte) im Januar 1990 von ISO übernommen wurde, fiel die Rationale weg. Sie gestattet interessante Einblicke in das Wie und Warum der Gestaltung des Standards (cf. »codifying existing practice«). In gedruckter Form beim o.g. Verlag erhältlich, kann man sie auch im Internet in Versionen als NONconforming Postscript oder LaTeX abrufen.

[Plau92] Plauger, P.J., The Standard C Library, Prentice-Hall, 1992
Eine Musterimplementierung der Standardbibliothek mit Quelltext, Anführung der entsprechenden Abschnitte des Standards, ausführlicher Beschreibung und Problemdiskussion. P.J. Plauger leitete den Ausschuß für die Standardisierung der C-Bibliothek beim X3J11 (ANSI) bzw. WG14 (ISO).

[Plau95] Plauger, P.J., and Jim Brodie, Standard C, A Reference, 3rd edition, Prentice-Hall, 1995
Jim Brodie (Softwarechef von Motorola) veranlaßte die Inangriffnahme der Standardisierung der Sprache C beim ANSI. Die Erweiterungen von 1994 sind in dieser Ausgabe schon berücksichtigt. Dies ist ein Nachschlagewerk, kein Lehrbuch. Es enthält eine DOS-formatierte Diskette mit dem ganzen Buch samt Index, Syntaxdiagrammen, Querverweisen etc. in aufbereitetem HTML-Format.

[Kern84] Kernighan, Brian W., and Rob Pike, The UNIX Programming Environment, Prentice-Hall, 1984
Die renommierten Meister der Unix- und C-Programmierung bieten hier die klassische Einführung in das Unix-System mit den besten, immer noch gültigen Beispielen zur Shell-Programmierung. Im Kapitel 7 wird eine Einführung in die sog. System Calls geboten, im Kapitel 8 ein kompletter Vierfunktionsrechner in C (K&R) entworfen, und inclusive Manual und Listing im Anhang dargeboten. Das Buch gibt es unter dem Titel »Der UNIX-Werkzeugkasten« auch in sehr guter deutscher Übersetzung (bei Hanser, 1987).

[Stev92] Stevens, W. Richard, Advanced Programming in the UNIX Environment, Addison-Wesley, 1992
Das Standardwerk zur systemnahen Programmierung unter allen Unix-Derivaten. Präzise und umfassend.

Index