

ANSI-C im Überblick

Peter Baeumle-Courth

Inhaltsverzeichnis

1 Vorwort	5
2 Einleitung	6
2.1 Geschichte und Einordnung der Programmiersprache C	6
2.2 Ein erstes Beispielprogramm	7
2.3 Reservierte Worte (Schlüsselwörter)	8
2.3.1 Befehlsschlüsselwörter	8
2.3.2 Schlüsselwörter für Speicherklassen	9
2.3.3 Schlüsselwörter für Datentypen	9
2.3.4 Weitere Schlüsselwörter	10
2.4 Grundlegender Programmaufbau	10
2.5 Kurzer Vergleich zwischen Pascal und C	11
3 Einfache Datentypen	13
3.1 Zeichen (char)	13
3.2 Numerische Datentypen	14
3.3 Konstanten (Literele)	15
3.4 Aufzählungstypen (enum)	16
3.5 Selbstdefinierte Datentypen (typedef)	16
3.6 Der Datentyp void	16
3.7 Typenkonversion (casting)	17
4 Operatoren und Funktionen	18
4.1 Operatoren	18
4.1.1 Arithmetische Operatoren	18
4.1.2 Inkrementoperatoren	18
4.1.3 Datentyp-Operatoren	19
4.1.4 Logische und Vergleichsoperatoren	19
4.1.5 Bit-Manipulationen	20
4.1.6 Zuweisungsoperatoren	20
4.1.7 Der Sequenzoperator	21
4.1.8 Der Bedingungsoperator	21

4.1.9 Übersichtstabelle: Prioritäten und Reihenfolge von Bewertungen	22
4.2 Der Preprocessor	23
4.2.1 Include-Direktive	23
4.2.2 Define-Direktive und Makros	23
4.2.3 Trigraphen	24
4.2.4 Bedingte Compilation	25
4.3 Funktionen	26
4.3.1 Formaler Aufbau einer Funktion	26
4.3.2 Parameter und Rückgabewerte	28
4.3.3 Bibliotheken und Headerfiles (Übersicht)	29
5 Terminal-Ein-/Ausgabe und Zeichenketten	30
5.1 Zeichenweise Ein- und Ausgabe	30
5.2 Zeichenketten (Strings)	31
5.3 Ein-/Ausgabe-Formatierung	33
5.3.1 Die Funktion printf()	33
5.3.2 Die Funktion scanf()	35
6 Kontrollstrukturen	37
6.1 Einzelanweisungen und Blöcke	37
6.2 Logische Ausdrücke und Verzweigung (if, if-else)	37
6.3 Iterationen (while, for, do-while)	38
6.3.1 Die while-Schleife	38
6.3.2 Die for-Schleife	38
6.3.3 Die do-while-Schleife	39
6.4 Mehrfachverzweigung (switch) und Marken	40
6.5 Abbruchmöglichkeiten (continue, break, return, exit)	40
6.6 Sprünge (goto)	42
7 Modularität, Gültigkeitsbereiche und Speicherklassen	43
7.1 Modularität und Gültigkeitsbereiche	43
7.2 Speicherklassen (auto, static, register, extern)	44
7.3 Attribute für Datentypen: const und volatile	45
7.3.1 Das Schlüsselwort const	45
7.3.2 Das Schlüsselwort volatile	45

8 Höhere Datentypen	46
8.1 Eindimensionale Arrays	46
8.2 Mehrdimensionale Arrays	47
8.3 Strukturen (struct)	48
8.4 Variante Strukturen (union)	49
8.5 Bit-Felder	51
8.6 Pointer	52
8.7 Pointer auf Pointer und Arrays von Pointern	54
8.8 Kommandozeilenargumente und Umgebungsinformation	56
8.8.1 Kommandozeilenargumente: argc und argv	56
8.8.2 Umgebungsinformation: envp	57
8.9 Rekursion und Rekursive Strukturen: Listen und Bäume	59
8.9.1 Rekursion	59
8.9.2 Lineare Listen	60
8.9.3 Bäume	63
9 Dateiverarbeitung	67
9.1 Vordefinierte Dateien stdin, stdout und stderr	67
9.2 Sequentieller Dateizugriff und I/O-Funktionen	68
9.2.1 Öffnen der Datei	68
9.2.2 Verarbeiten der Datensätze	69
9.2.3 Schließen der Datei	70
9.2.4 Beispiele zur sequentiellen Dateiarbeit	71
9.3 Wahlfreier Zugriff (random access)	73
10 Ergänzungen	76
10.1 make und Makefiles	76
10.2 Debugger	80
10.3 Profiler	83
10.4 Cross-Reference-Listen mit cxref	85
11 Literaturhinweise	89
12 Index	91

1 Vorwort

Die nachfolgende Ausarbeitung zur Programmiersprache C ist als Begleitung des Unterrichts gedacht; daher erhebt es keinen Anspruch auf Vollständigkeit hinsichtlich des Umfangs der standardisierten Programmiersprache ANSI¹-C. Ebenso enthält dieses Script bewußt nur wenige, meist kurze, auf relevante Details reduzierte Beispielprogramme oder Programmauszüge und keinerlei Übungsaufgaben.

C ist ganz offensichtlich keine gut geeignete Lehrsprache, denn es wird sich bereits an frühen Programmbeispielen zeigen, daß sehr schnell die verschiedensten Aspekte (dynamische Speicherverwaltung, Pointer, Preprocessor-Konzept und Bibliotheken, Ausgabeformatierung mit der printf()-Routine) auftauchen.

Wesentliche Teile dieses Scripts sind angelehnt an das im Literaturverzeichnis aufgeführte, grundlegende Standardwerk von Kernighan und Ritchie sowie an Unterrichtsmaterialien und Ausarbeitungen meiner Kolleginnen und Kollegen.

Für das Verständnis werden solide Kenntnisse der Programmierung in Pascal vorausgesetzt; an zahlreichen Stellen werden Sachverhalte auch durch einen Vergleich mit Pascal zu erläutern versucht. In einem gesondert Abschnitt werden insbesondere exemplarische Teile von Pascal und C einander gegenübergestellt.

In den Ergänzungen wird auf einige der üblichen Zusatzprogramme zur Softwareentwicklung (Makefiles, Debugger, Profiler, Cross-Reference-Lister) eingegangen; aus Platzgründen konnte auf zahlreiche andere nicht eingegangen werden: z.B. den Syntaxprüfer lint (siehe hierzu die Online-Hilfe unter UNIX), die zum Bereich des Compilerbaus gehörigen Tools lex und yacc oder auch die verschiedenen Versionskontrollsysteme (wie etwa das SCCS unterUNIX).

Noch ein Hinweis für mittellose DOS-Anwender: wer auch gerne unter DOS C erproben möchte, jedoch kein Geld für einen der kommerziellen C-Compiler hat oder ausgeben möchte, kann im PC-Bereich die älteren Versionen der Borland-C- und -C++-Compiler sehr preisgünstig erwerben.

¹ANSI - American National Standards Institute, die US-amerikanische Normungsbehörde, entspricht in etwa der DIN in Deutschland.

2 Einleitung

2.1. Geschichte und Einordnung der Programmiersprache C

C ist eine *all purpose language*, eine Programmiersprache für sehr viele verschiedene Anwendungsbereiche. Bereits in einer Statistik über mehrere Tausend Stellenangebote in Deutschland im Jahre 1990 hatte C die allseits unwahrscheinlich beliebte Programmiersprache COBOL überrundet, der Trend geht seitdem ungebrochen stark in Richtung C sowie der von Bjarne Stroustrup hervorgebrachten objektorientierten Erweiterung, C++.

C ist sehr eng mit dem Betriebssystem UNIX verbunden, für das die Sprache sogar eigens entwickelt worden ist. Der Unterricht wird demzufolge naturgemäß auf der Grundlage des Betriebssystems UNIX stattfinden.

Wichtige Ideen für C entstammen den beiden Programmiersprachen BCPL, die Martin Richards, und B, die Ken Thompson 1970 für das erste UNIX-System auf einer DEC-Anlage entwickelt hat. C ist eine typisierte Sprache, wobei die Datentypen jedoch bei weitem nicht so streng überwacht werden wie bei Pascal. Weiterhin finden sich die klassischen Datenstrukturen wie Felder, Zeiger, Strukturen (records) und Variante Strukturen (unions); C verfügt über die aus Pascal bekannten Kontrollstrukturen: Blockung (mit geschweiften Klammern { und }), Alternativen (if, if-else, switch) und Schleifen (while, for, do-while). Daneben gibt es Möglichkeiten des vorzeitigen Verlassens einer Kontrollstruktur (z.B. mit break). Funktionen können Ergebniswerte beliebiger Datentypen besitzen und auch rekursiv aufgerufen werden. (Viel Spaß.) Im Gegensatz zu Pascal können in C Funktionen jedoch nicht verschachtelt definiert werden. Variablen können lokal in einem Block, in einer Funktion, in einer Datei oder in einem gesamten Programm, das aus mehreren Quelltextdateien besteht, verfügbar sein. Lokale Variablen sind defaultmäßig *automatic*, was bedeutet, daß sie bei Betreten des Gültigkeitsbereiches neu angelegt und bei Verlassen desselben wieder zerstört werden.

C verwendet einen *Preprocessor*, der Makros im Quelltext ersetzt, andere Quelldateien (via #include) einfügt und bedingte Compilation erlaubt. Auch durch dieses Konzept kann C sehr leicht auf sich ändernde Verhältnisse, z.B. andere Systeme, angepaßt werden.

Der eigentliche Kern der Programmiersprache C ist sehr klein. Er umfaßt zum Beispiel noch nicht einmal Routinen für Terminal-Ein- und -Ausgabe! Die Leistungsfähigkeit erhält C durch die sogenannten Bibliotheken (*Libraries*), die jeweils konkret eingebunden werden müssen.

C ist einerseits sehr maschinennah, beispielsweise stehen Operationen zur bitweisen Verarbeitung zur Verfügung und über Zeiger sind Speicheradressen direkt ansprechbar. Andererseits ist C sehr portabel, wenn man sich nur an die allgemeinen Richtlinien von ANSI C hält. (1989 wurde der Standard vom amerikanischen Normungsinstitut ANSI verabschiedet.)

Durch das liberale Umgehen mit z.B. Datentypen und die vielfältigen Möglichkeiten, sehr kompakten (sprich: schwer lesbaren) Code zu schreiben, wird C oft nachgesagt, daß es extrem fehleranfällig sei, die Programme schwerer als babylonische Keilschriften zu entziffern seien und Hackermentalitäten begünstigt würden. Kernighan und Ritchie betonen daher, daß C auf

der grundlegenden Annahme basiert, daß Programmierer wissen, was sie tun, und daß sie ihre Absichten eben nur explizit kundtun müßten.

2.2. Ein erstes Beispielprogramm

Das minimale, leere C-Programm sieht aus wie folgt.

```
main()
{
}
```

Trotzdem erkennt man schon einiges:

- Das Hauptprogramm ist (nur) eine Funktion mit dem festgelegten Namen *main*.
- Hinter dem Funktionsnamen folgt die formale Parameterliste, hier ist sie leer, daher stehen die runden Klammern einfach so einsam umher.
- Der Anweisungsteil der Funktion (des Hauptprogramms) ist ebenfalls leer; zusammengehalten werden die Anweisungen durch das in erster Näherung dem Pascal-begin-end entsprechenden geschweiften Klammerpaar { und }.

Etwas interessanter wird dann schon das zweite Beispiel: das Standardprogramm "*Hello, world!*".

```
/* hello.c */

#include <stdio.h>

void main(void)
{
    printf("Hello, world!\n");
} /* end main */
```

Nun ist schon mehr zu erkennen:

- In der ersten Zeile steht ein Kommentar. Kommentare, die über eine beliebige Strecke im Quelltext gehen können, beginnen mit den Zeichen `/*` und enden mit `*/`.
- Die nächste Direktive (`#include`) geht an den Preprocessor: er wird hier aufgefordert, die gesamten Deklarationen der sogenannten Headerdatei einzubinden. (Dateinamen von Headerfiles enden üblicherweise auf `.h`.)
- Vor dem Funktionsnamen `main` ist nun das Wort `void` zu lesen. Dieser seltsame Datentypname steht nur dafür, daß diese Funktion nichts zurückliefern soll. Im Prinzip wurde damit einem Pascal-Programmierer gesagt, daß `main` hier wie eine Prozedur verwendet wird.
- Innerhalb der formalen Parameterliste steht nun nicht mehr nichts, sondern das Wörtchen `void` um anzudeuten, daß da nichts stehen soll. So macht das Spaß. Auch wenn das paradox anmutet: so signalisiert ein Programmierer, daß die Funktion (hier: `main`) tatsächlich keine Parameter erhalten soll.
- Der Funktionsname `printf` ist das write von C. Wie oben bereits gesagt wurde, ist die Funktion `printf` nicht im eigentlichen C enthalten, sondern eine Bibliotheksfunktion. Über

die obige Direktive `#include <stdio.h>` wurde gesagt, daß die Standard-Ein- und -Ausgabe-Routinen (standard input output) mit eingebunden werden sollen.

- f) Die Funktion `printf` hat hier einen Parameter: eine Zeichenkette, die in doppelten Hochkommata notiert wird. Darin steht zum einen der Klartext "Hello, world!", zum anderen folgt danach jedoch noch etwas Besonderes: `"\n"` ist eines von mehreren zur Verfügung stehenden Ersatzsymbolen, `"\n"` steht speziell für einen Zeilenvorschub.
- g) Das Semikolon hinter dem Aufruf von `printf` muß übrigens sein, anders als bei entsprechenden Situationen in Pascal. Das Semikolon in C schließt (u.a.) eine Anweisung ab! Nachstehend noch der Vergleich zum funktionsäquivalenten Pascal-Programm.

```
{ hello.p bzw. hello.pas }  
program HelloWorld(output);  
begin  
  writeln('Hello, world!')  
end.
```

2.3. Reservierte Worte (Schlüsselwörter)

Die Programmiersprache ANSI C kennt eine Reihe reservierter Worte (Schlüsselwörter), die in den nächsten Abschnitten sortiert aufgelistet werden sollen.

2.3.1 Befehlsschlüsselwörter

Die folgenden Schlüsselwörter stehen für Befehle der Sprache C.

```
break case continue default do else (entry)  
for (goto) if return sizeof switch while
```

Anmerkungen:

- a) Das Schlüsselwort "entry" ist ein Überbleibsel aus den Pioniertagen von C; es ist in ANSI C nicht inhaltlich implementiert.
- b) Das Schlüsselwort "goto" existiert leider auch noch; mit ihm werden die im Bereich der strukturierten Programmierung als unmoralisch gebrandmarkten Sprünge gekennzeichnet.
- c) Das Schlüsselwort "sizeof" steht streng genommen nicht für einen Befehl, sondern für einen Operator. Dieser liefert die Größe (in Bytes) einer Variablen oder eines Datentyps zurück.

2.3.2 Schlüsselwörter für Speicherklassen

Auf Speicherklassen wird in einem späteren Kapitel eingegangen. Hier nur bereits übersichtsartig die dazugehörenden Schlüsselwörter.

- a) *auto* kennzeichnet eine Variable, deren Allokation beim Eintritt in den Gültigkeitsbereich und Deallokation beim Austritt daraus erfolgt;
- b) *extern* sind globale Variablen, die für die gesamte Programmlaufzeit allokiert werden;
- c) *register* entspricht *auto*, nach Möglichkeit wird jedoch statt eines Speicherplatzes ein Register belegt; bei modernen Compilern ist dieses Schlüsselwort relativ obsolet, da bei der Codegenerierung in der Regel sowieso optimiert wird.
- d) *static* kennzeichnet Variablen, deren Allokation für die gesamte Programmlaufzeit erfolgt, deren Gültigkeit jedoch block- oder modullokal festgelegt ist.

2.3.3 Schlüsselwörter für Datentypen

Für verschiedene vordefinierte Datentypen stellt C reservierte Worte bereit.

- a) *char*: Zeichen (1 Byte)
- b) *double*: Gleitkommazahl, doppelte Genauigkeit
- c) *enum*: Kennzeichnung für Aufzählungstyp
- d) *float*: Gleitkommazahl
- e) *int*: Ganzzahl
- f) *long* (oder *long int*): Ganzzahldatentyp
- g) *unsigned*: Kennzeichnung zur Interpretation eines Datentyps ("ohne Vorzeichen"), z.B. "unsigned int" oder "unsigned char"
- h) *short*: Ganzzahl
- i) *signed* Kennzeichnung zur Interpretation eines Datentyps als vorzeichenbehaftet, z.B. *signed int* (=int) oder *signed char*
- j) *struct*: Struktur (=Record)
- k) *typedef*: Eigendefinition von Datentypen
- l) *union*: Variante Struktur (=varianter Record)
- m) *void*: "leerer" Datentyp

2.3.4 Weitere Schlüsselwörter

Daneben gibt es noch zwei weitere Schlüsselwörter:

- a) *const*: kennzeichnet einen Datentyp für einen nicht veränderbaren Speicherplatz.
- b) *volatile*: Typattribut für eine Variable, die durch externe Einflüsse (von außerhalb des Programmes) verändert werden kann, beispielsweise die Systemuhr.

2.4. Grundlegender Programmaufbau

Ein C-Programm besteht generell aus den folgenden Komponenten:

- a) Den Preprocessor-`#include`-Anweisungen, mit denen externe Dateien (z.B. die bereits erwähnten Headerfiles) in den Quelltext einbezogen und Schnittstellen zu den Bibliotheksroutinen der C Library geschaffen werden.
- b) Den Preprocessor-`#define`-Anweisungen, mit denen symbolische Konstante und Makros definiert werden können.
(Beispiel: `#define MAXIMUM 100`)
- c) Datentypen und Variablen, die vor dem Hauptprogramm deklariert werden; hier werden globale bzw. externe Variablen und globale Datentypen festgelegt, die allen Teilen des C-Programms zugänglich sein können.
- d) Das Hauptprogramm `main` (genau genommen: die Funktion `main()`): hier findet der Programmeinstieg statt. `main()` darf und muß in jedem C-Programm genau einmal vorkommen; häufig steht es als erste Funktion in dem Quelltext, sofern das Programm nicht sowieso mehrere Quelltextdateien umfaßt, bei denen dann `main()` oft eine eigene Datei (z.B. `main.c`) spendiert bekommt.
- e) Es folgen dann ggf. weitere Funktionen.

Dabei ist es im Prinzip gleichgültig, ob `main()` die erste oder die letzte Funktion oder irgendeine zwischen den anderen Funktionen ist; es ist lediglich guter Stil, `main()` entweder stets am Anfang oder stets am Ende eines Ein-Datei-Programmes zu plazieren.

2.5. Kurzer Vergleich zwischen Pascal und C

Nachfolgend sollen übersichtsartig (für einen ersten Eindruck und zum späteren Nachschlagen) einige Dinge von Pascal und C gegenüber gestellt werden. (Diese Aufstellung wurde dem Buch von Müldner und Steele entlehnt.) Auf die einzelnen Details wird später in diesem Script bzw. im Unterricht näher eingegangen.

<u>ANSI-C</u>		<u>Pascal</u>
<code>/* ... */</code>	Kommentare	<code>{ ... }</code>
<code>short, int, long char float, double</code>	Datentypen	<code>integer char real</code>
<code>#define PI 3.14 const float PI=3.14;</code>	Konstanten	<code>const PI=3.14;</code>
<code>int i;</code>	Variablen	<code>var i: integer;</code>
<code>void main(void) { } }</code>	Hauptprogramm	<code>program xy(input,output); begin end.</code>
<code>+ - * / + -</code>	Grundrechenarten Vorzeichen	<code>+ - * / + -</code>
<code>/ %</code>	Ganzzahldivision	<code>div, mod</code>
<code>scanf("%d",&i); printf("%d",i);</code>	Eingabe Ausgabe	<code>read(i) write(i)</code>
<code>< <= > >= == !=</code>	Vergleichsoperatoren	<code>< <= > >= = <></code>
<code>&& !</code>	logische Operatoren	<code>and, or, not</code>
<code>{ ... }</code>	Verbundanweisung (Block)	<code>begin ... end</code>
<code>if (x<1) y=2; else y=3;</code>	if-Konstrukt (Alternative)	<code>if x<1 then y:=2 else y:=3;</code>
<code>while (x<100) x += 1;</code>	kopfgesteuerte Schleife	<code>while x < 100 do x := x + 1</code>
<code>do { ... } while (x<100);</code>	fußgesteuerte Schleife	<code>repeat ... until x>=100</code>
<code>for (i=1; i<=100; i++) </code>	for-Schleife (entsprechen sich nicht ganz)	<code>for i:=1 to 100 do </code>

<pre>switch(i) { case 1: printf("eins"); break; case 2: printf("zwei"); break; default: ... }</pre>	Mehrfachauswahl	<pre>case i of 1: write('eins'); 2: write('zwei'); otherwise { oder else } ... end</pre>
<pre>char a[100]; char b[6][11];</pre>	Felder	<pre>var a: array[0..99] of char; var b: array[0..5,0..10] of char;</pre>
<pre>struct { float x, y; } r;</pre>	Strukturen	<pre>var r: record x, y: real; end;</pre>
<pre>typedef enum { ROT, GRUEN, BLAU } farbe;</pre>	Aufzählungstyp	<pre>type farbe= (rot,gruen,blau);</pre>

3 Einfache Datentypen

C kennt im wesentlichen die von Pascal gewohnten einfachen Datentypen - mit Ausnahme von boolean. Eine Variable wird deklariert in der syntaktischen Form

```
typename variablenname;
```

Der Operator sizeof dient zur Feststellung, wieviele Bytes ein Datentyp oder eine Variable dieses Datentyps benötigen. Beispiele hierzu finden Sie nachfolgend.

3.1. Zeichen (char)

Eine Variable vom Datentyp char benötigt 8 Bits (1 Byte) Speicherplatz und kann jeweils ein Zeichen aus dem der Maschine zugrundeliegenden Code (bei uns i.d.R. ASCII²) aufnehmen.

3.1.1 Beispiel

```
void main(void)
{
    char zeichen;          /* Die Variable zeichen wird */
    zeichen = 'A';        /* vom Datentyp char        */
                          /* deklariert und erhält per */
                          /* Zuweisung den Wert 'A'.  */
}
```

Die Variable zeichen kann jeweils eines der 256 Zeichen des (in der Regel ASCII-)Codes aufnehmen. C unterscheidet zwei Varianten: signed bedeutet, daß der Datentyp char wie der numerische Bereich -128..127 behandelt wird, das ASCII-Zeichen 255 wird also (numerisch) als -1 interpretiert; demgegenüber bedeutet unsigned, daß char wie der Bereich 0..255 verwendet wird, das ASCII-Zeichen 255 wird also auch numerisch als 255 interpretiert. Welche Interpretation ein konkreter Compiler vornimmt, ist maschinenabhängig, aber auch vom Programmierer oder von der Programmiererin einstellbar. Auf dem HP9000-System von Hewlett-Packard ist char beispielsweise als signed voreingestellt. Aus diesem Grund wird in C häufig, z.B. bei der Bibliotheksfunktion getch(), der (stets vorzeichenbehaftete) Datentyp int (statt char) für ein Zeichen verwendet! Diese Betrachtung mag einem treuen Pascalianer seltsam vorkommen; wir werden aber sehr schnell sehen, daß C es anscheinend nicht so genau nimmt mit der Unterscheidung von char und numerischen (ganzzahligen) Datentypen: in Wahrheit ist char jedoch nichts anderes als ein numerischer Datentyp, dessen Ausprägungen lediglich bedarfsweise als Repräsentanten des dem Rechner zugrundeliegenden Codes (z.B. ASCII) interpretiert werden!

Will man von der jeweiligen Voreinstellung abweichen, kann man eine Variable explizit als signed char oder unsigned char vereinbaren.

²ASCII - American Standard Code for Information Interchange

3.2. Numerische Datentypen

Der Datentyp `int` (integer) vertritt den (vorzeichenbehafteten) Ganzzahlbereich, entspricht also auch der Angabe `signed int`. Der Speicherplatzbedarf ist von ANSI nicht vorgeschrieben, also maschinenabhängig; bei HP-UX 9.0 beträgt dieser 4 Bytes, bei PC-Compilern in der Regel 2 Bytes. In der unten auszugsweise abgedruckten Headerdatei `limits.h` (hier in der Version von Hewlett Packards UNIX-Derivat HP-UX) werden symbolische Konstanten bereitgestellt, z.B. `INT_MAX`, dem größten Wert aus dem Bereich des Datentyps `int`. (Dies entspricht dem `MAXINT` von Pascal.)

```
/* /usr/include/limits.h .. stark gekürzt .. */
#define CHAR_BIT 8 /* Number of bits in a char */
#define CHAR_MAX 127 /* Max integer value of a char */
#define CHAR_MIN (-128)/* Min integer value of a char */
#define INT_MAX 2147483647 /* Max decimal value of an int */
#define INT_MIN (-2147483648)/* Min decimal value of an int */
```

Mit `unsigned int` kann erwartungsgemäß angegeben werden, daß der Speicherplatz ohne Vorzeichenbit interpretiert wird, der Wertebereich also bei 0 (statt bei `INT_MIN`) beginnt. Statt der Deklaration `unsigned int i`; genügt im übrigen bereits `unsigned i`;

`short` und `long` (`int`) sind weitere Ganzzahldatentypen, jeweils defaultmäßig als `signed` interpretiert. Die Größen für diese Datentypen sind wiederum maschinenabhängig, bei HP-UX 9.0 wie bei den gängigen PC-Compilern belegt ein `short`-Speicherplatz 2 Bytes und einer vom Typ `long` (`int`) 4 Bytes. Analog zu `unsigned int` kann auch hier über das vorangestellte Wort `unsigned` eine entsprechende andere Interpretation erzwungen werden.

Beispiel:

```
void main(void)
{
    unsigned short us;
    short ss; /* = signed short */
    unsigned long ul;
    long sl; /* = signed long */
}
```

Die ganzzahligen Datentypen und der Datentyp `char` (sowie die später erläuterten Aufzählungstypen) werden (entsprechend wie in Pascal) auch als `ordinales` oder, etwas irreführend vielleicht, als `integer-Datentypen` bezeichnet.

Mit `float`, `double` und `long double` stehen drei verschiedene Gleitkommatypen zur Verfügung. Wieder sind die Bereiche maschinenabhängig. Zu den Gleitkommatypen stehen in der Datei `float.h` (sh. unten) entsprechende vordefinierte Konstanten.

```
/* /usr/include/float.h .. stark gekürzt .. */
#define FLT_MAX 3.40282347E+38
#define FLT_MAX_10_EXP 38
#define DBL_MAX 1.7976931348623157E+308
#define DBL_MAX_10_EXP 308
```

3.3. Konstanten (Literele)

Eine ganzzahlige Konstante wie 123 hat den Typ int; eine long-Konstante wird (zur Betonung) mit der Endung l oder L geschrieben: 123L ist damit vom Typ long³. Ist eine ganzzahlige Konstante zu groß für int, so wird sie jedoch implizit als long interpretiert. Für vorzeichenlose (unsigned) Konstanten kann das Suffix U (oder u) verwendet werden: 123U ist eine Konstante vom Typ unsigned int. Entsprechend ist 123UL eine Konstante vom Typ unsigned long.

Ganzzahlkonstanten können auch oktal und hexadezimal angegeben werden. (Freude kommt auf!) Beginnt eine ganzzahlige Konstante mit einer 0, so wird sie als Oktalzahl interpretiert: 007 ist dezimal 7, 010 ist dezimal 8. Beginnt sie dagegen mit 0x oder 0X, so wird sie hexadezimal interpretiert: 0x1f oder 0x1F stehen für den dezimalen Wert $1 \cdot 16 + 15 \cdot 1 = 31$.

Gleitpunktkonstanten enthalten einen Dezimalpunkt (123.45) und/oder einen Exponenten (12E-2 steht für 0.12, 1.234E2 steht für 123.4). Ohne ein spezielles Suffix sind diese vom Typ double, mit dem Suffix F (oder f) float, mit L vom Typ long double!

Eine Zeichenkonstante (z.B. 'A') wird stets als ganzzahlig angesehen. So ist etwa im ASCII-Code 'A' der Wert 65, '0' ist 48. Häufig werden Zeichenkonstanten zum Vergleich mit anderen Zeichen herangezogen.

Gewisse Sonderzeichen können auch in sogenannten Ersatzdarstellungen angegeben werden: der Zeilenvorschub (*LineFeed*) oder der Wagenrücklauf (*Carriage Return*) können als '\r' bzw. '\n' (auch als *NewLine* gelesen) geschrieben werden. Daneben kann in der Form '\0???' bzw. '\x???' ein Zeichen oktal oder hexadezimal angegeben werden⁴.

Beispiel: '\007' und '\x7' stehen gleichermaßen für das ASCII-Zeichen Nr. 7 (Klingelzeichen).

Hier die Ersatzdarstellungen im Überblick:

- \a Klingelzeichen (*Bell*)
- \b Backspace
- \f Seitenvorschub (*FormFeed*)
- \n Zeilenvorschub (*LineFeed*)
- \r Wagenrücklauf (*Carriage Return*)
- \t Tabulatorzeichen
- \v Vertikaler Tabulator
- \\ steht für den Backslash \
- \? Fragezeichen
- \' Anführungszeichen
- \" doppeltes Anführungszeichen

Warnung: Bitte unterscheiden Sie künftig zwischen 'x' und "x"! Das einfache Hochkomma steht in C für ein einzelnes Zeichen, das doppelte Hochkomma wird jedoch für Zeichenketten (Strings) stehen!

3.4. Aufzählungstypen (enum)

³In der Regel wird man das große L verwenden, da ein kleines l von einer 1 nicht immer leicht zu unterscheiden ist.

⁴Beachten Sie bitte: trotz der mehreren zu tippenden Zeichen stehen hier einfache Hochkommata, denn das gesamte Gebilde steht in C für ein einzelnes Zeichen!

C kennt wie Pascal Aufzählungstypen (enumerated types), auch wenn diese in C seltener eingesetzt werden. Mit der Deklaration

```
enum boolean { FALSE, TRUE };
```

wird ein Datentyp (enum) boolean vereinbart; implizit sind damit die (letztlich numerischen) Konstanten FALSE (=0) und TRUE (=1) vereinbart worden⁵. Allerdings hat C aufgrund seiner äußerst liberalen Lebenseinstellung keine Probleme damit, einer in diesem Sinne deklarierten boolean-Variablen auch Werte wie 2, 17 oder 'A' zuzuweisen!

Zwei Beispiele solcher enum-Deklarationen:

```
enum wochentage { MO=1, DI, MI, DO, FR, SA, SO };
```

Das geht auch: MO wird damit auf 1 statt auf 0 festgelegt; alle anderen Werte folgen: DI=2 usw.

```
enum wochentage tag;      /* Deklaration einer Variablen*/
tag=DI;                  /* und Wertzuweisung          */
```

```
enum faecher { BIOLOGIE, CHEMIE, MATHEMATIK, PHYSIK } fach;
/* ... */
for (fach=BIOLOGIE; fach<=PHYSIK; fach++)
    /* tueirgendwas */
```

3.5. Selbstdefinierte Datentypen (typedef)

Wiederum wie bei Pascal können auch in C eigene Typ(nam)en geschaffen werden. Die Syntax hierfür ist einfach: typedef <alter-Typ> <neuer-Typname>; Mit der Deklaration

```
typedef int INTEGER;
```

wird ein Datentyp namens INTEGER geschaffen, der synonym mit int verwendet wird. Die Verwendung von typedef wird später sinnvoller werden bei höheren und strukturierten Datentypen.

3.6. Der Datentyp void

Einen etwas eigenwilligen Datentypen kennt (ANSI) C unter dem Namen *void*. Hiermit kann explizit gesagt werden, daß eine Funktion keinen Rückgabewert besitzt, oder daß eine Parameterliste leer ist. Das haben wir bereits im Eingangsbeispiel hello.c gesehen.

Eine Variable kann (natürlich) nicht vom Datentyp void sein: der Compiler meldet dann "*unknown size for ...*", denn der Datentyp void hat keine Größe!

3.7. Typenkonversion (casting)

⁵Vgl. hierzu den Abschnitt zu logischen Ausdrücken.

Hat ein mehrwertiger Operator mit Werten oder Variablen von verschiedenen Typen zu tun, so wird (häufig) eine implizite oder explizite Typkonversion (sogenanntes *Casting*) durchgeführt.

Hierzu ein kleines konkretes Beispielprogramm.

```
/* casting.c */
void main(void)
{
    char c;
    int i;
    float x;

    i=65;
    c=i;          /* funktioniert, c=65='A'          */
    c=i+32;      /* geht auch, c=97='a'          */

    i=7/9*c;     /* geht ebenfalls, ist aber 0,      */
                /* denn 7/9 ist (als int!) 0 !!!   */

    x=3.45;
    i=x;        /* geht, i wird auf 3 gesetzt      */
    x=i;        /* geht natürlich auch            */
}
```

Neben den impliziten Typumwandlungen gibt es auch explizite, sogenannte Casts oder Castings⁶: mit der Syntax (<typename>) expression wird der entsprechende Ausdruck auf den angegebenen Datentyp projiziert.

Ein Beispiel:

```
float x;
x= 7/9;          /* damit wird x auf 0 gesetzt!      */
x= (float)7/9;  /* damit wird x auf 0,777778 gesetzt! */
```

⁶zu deutsch: Gipsverband

4 Operatoren und Funktionen

In diesem Kapitel geht es um eines der Herzstücke von C: mit den Operatoren und den Funktionen wird die Umsetzung der verschiedenen Algorithmen in die Programmiersprache C erst möglich.

4.1. Operatoren

Im folgenden sollen kurz sämtliche Operatoren⁷ von C vorgestellt werden. Einige davon werden naturgemäß erst zu einem späteren Zeitpunkt verständlich werden.

4.1.1 Arithmetische Operatoren

C besitzt für die vier Grundrechenarten die entsprechenden zweistelligen (binären) arithmetischen Operatoren + (Addition), - (Subtraktion), * (Multiplikation) und / (Division), die jeweils für alle numerischen Datentypen existieren.

Der Ergebnistyp⁸ einer solchen Operation richtet sich dabei nach den Operanden; so ist $7/9$ (als int) 0, $7.0/9$ jedoch der erwartete Wert $0,777778!$

Daneben gibt es den Modulo-Operator %, der den Rest bei der Ganzzahldivision in C darstellt ($13\%5=3$) und dem mod aus Pascal entspricht. Der entsprechende div-Operator ist der /, denn $13/5=2$.

4.1.2 Inkrementoperatoren

In C können die häufig gebrauchten Formulierungen, die in Pascal noch $i:=i+1$ lauteten, kürzer gefaßt werden mit den sogenannten Inkrementoperatoren⁹: $i++$ oder $++i$. Anstelle der C-Anweisungen $i=i-1$ kann geschrieben werden $i--$ oder $--i$ (Inkrementoperatoren). Die Operatoren $++$ und $--$ (jeweils als Präfix- oder Suffixoperator) inkrementieren bzw. dekrementieren die betreffende Variable jeweils um 1.

4.1.3 Datentyp-Operatoren

⁷Unter einem *Operator* versteht man traditionell primär Rechenvorschriften wie + und -; im Rahmen der Programmierung sind Operatoren aber weiter gefaßt ähnlich wie Funktionen zu verstehen, nur daß die Operatoren mit einer anderen Syntax aufgerufen werden und in klassischen Programmiersprachen wie C oder Pascal nicht für eigene Datentypen neu definiert werden können. Dies ist in der objektorientierten Erweiterung C++ dagegen möglich.

⁸Generell liefert eine arithmetische (Grundrechen-)Operation, die auf zwei Ganzzahloperanden wirkt, eine Ganzzahl zurück; ist mindestens ein Gleitkomma-Operand beteiligt, so liegt das Resultat in dem entsprechenden Gleitkommaformat vor.

⁹Der Unterschied zwischen $i++$ und $++i$ wird später deutlich werden.

Wie bereits erwähnt: sizeof ist ein Operator, mit dem die Speicherplatzanforderungen einer Variablen oder eines Datentyps abgefragt werden können. Daneben ist der cast-Operator noch zu erwähnen, der eine explizite Typumwandlung erzwingt.

Beispiel:

```
unsigned long i;
int memory;
memory=sizeof i;          /* Speicherbedarf von i          */
memory=sizeof(unsigned long); /* Speicherbedarf von uns.long */
i = (unsigned long) memory; /* i wird der gecastete Wert   */
                          /* von memory zugewiesen      */
```

4.1.4 Logische und Vergleichsoperatoren

In C gibt es keinen Datentyp boolean wie in Pascal. Für C ist jeder numerische Wert ungleich 0 gleichwertig mit TRUE (wahr), nur die 0 wird als FALSE (falsch) interpretiert. Dementsprechend gibt es auch logische Operatoren, die als Ergebnisse die Werte 0 oder "ungleich 0", meistens konkret den Wert 1, zurückliefern. So ist && der logische UND-Operator, || der logische ODER-Operator und ! kennzeichnet die (logische) Negation.

Achtung: Verwechseln Sie bitte die logischen nicht mit den bitweisen Operatoren, die im Abschnitt Bit-Manipulationen vorgestellt werden!

Die üblichen sechs Vergleichsoperatoren besitzt C ebenfalls: mit < wird auf "kleiner als" geprüft, mit > auf "größer als", mit <= bzw. >= auf "kleiner oder gleich" bzw. "größer oder gleich", mit == wird die Gleichheit geprüft und mit != die Ungleichheit.

Achtung: C ist immer noch sehr liberal! Wird versehentlich a=b statt a==b geschrieben, so stört das den C-Compiler nicht; statt der logischen Abfrage a==b "ist a gleich b?" wird jedoch bei "a=b" der Wert von b der Variablen a zugewiesen und dieser Wert dient als Rückgabewert und somit als Beurteilung, ob TRUE oder FALSE vorliegt; nur wenn b den Wert 0 hat, ist "a=b" FALSE, sonst stets TRUE! Dies ist eine sehr häufige Fehlerquelle!!!

4.1.5 Bit-Manipulationen

Speziell für die ordinalen Datentypen (char, short, int, long in beiden Varianten signed oder unsigned) existieren sechs Operatoren für sogenannte Bit-Manipulationen.

Operator	Wertigkeit	Bezeichnung / Erläuterung
&	binär	bitweise Und-Verknüpfung
	binär	bitweise Oder-Verknüpfung
^	binär	exclusive Oder-Verknüpfung (XOR)
<<	binär	Bit-Verschiebung nach links (shift left)
>>	binär	Bit-Verschiebung nach rechts (shift right)
~	unär	bitweises Komplement

Ein kleines Beispiel hierzu:

```
unsigned char a,b,c;      /* Bitnummer: 7654 3210      */
a=0x11;                  /* = 17      Bitmuster: 0001 0001      */
b=0x0F;                  /* = 15      0000 1111      */

c=a & b;                 /* c wird gesetzt auf: 0000 0001      */
c=a | b;                 /* c wird gesetzt auf: 0001 1111      */
c=a ^ b;                 /* c wird gesetzt auf: 0001 1110      */
c=a << 1;                /* c wird gesetzt auf: 0010 0010      */
c=b >> 2;                /* c wird gesetzt auf: 0000 0011      */
c=~a;                   /* c wird gesetzt auf: 1110 1110      */
```

4.1.6 Zuweisungsoperatoren

C kennt eine Reihe von Zuweisungsoperatoren. Während Pascal nur den Operator := für die direkte Zuweisung kennt, gibt es bei C die folgenden. Für die Beispiele seien die Deklarationen int a,b,c; zugrundegelegt.

```
a = b + c;   /* gewöhnliche Zuweisung      */
a += b;     /* steht für a = a + b;      */
a -= b;     /* steht für a = a - b;      */
a *= b;     /* steht für a = a * b;      */
a /= b;     /* steht für a = a / b;      */
a %= 5;     /* a = a % 5;                */
a &= b;     /* a = a & b;                */
a |= b;     /* a = a | b;                */
a ^= b;     /* a = a ^ b;                */
a <<= 2;    /* a = a << 2;               */
b >>= a;    /* b = b >> a;               */
```

4.1.7 Der Sequenzoperator

Mit dem Sequenzoperator , können mehrere Anweisungen zu einer einzigen zusammengefaßt werden. Dieser wird später z.B. innerhalb der for-Schleife gelegentlich verwendet.

Beispiel:

```
int i=0,j=1,k=2;
    /* Eine "horizontale" Sequenz von drei Anweisungen */
i=1, j*=i, k+=i;
```

4.1.8 Der Bedingungsoperator

In dem Kapitel Kontrollstrukturen werden die if- und anderen Verzweigungsstrukture von C behandelt. Im Reigen der Operatoren findet sich ein einziger dreiwertiger (*ternärer*) Operator, der Fragezeichenoperator oder Bedingungsoperator.

An die Stelle eines beliebigen Ausdruckes kann auch ein Ausdruck der Form <bedingung> ? <ausdruck1> : <ausdruck2> treten. Trifft die <bedingung> zu, d.h. ist <bedingung> != 0, so wird <ausdruck1> genommen, andernfalls <ausdruck2>.

Beispiel:

```
a = (b > c ? b : c);
```

Hier wird a der Wert von b zugewiesen, falls b > c ist; andernfalls wird a der Wert von c zugewiesen.

4.1.9 Übersichtstabelle: Prioritäten und Reihenfolge von Bewertungen

Nachstehend werden die Prioritäten und die Bewertungsreihenfolgen, die sogenannten Assoziativitäten, der Operatoren in ANSI-C aufgelistet. Die Prioritäten sind von oben nach unten abnehmend aufgeführt; die Operatoren innerhalb einer Zeile werden gemäß ihrer Assoziativität verarbeitet. Der * unter Priorität 14 ist die Pointerreferenzierung, der unter 13 ist der Rechenoperator Multiplikation, die Zeichen + und - unter 14 sind die unären Vorzeichenoperatoren, das &-Zeichen unter Priorität 14 ist der Adreßoperator, das &-Zeichen unter 8 ist das bitweise Und.

Priorität	Operator	Assoziativität
15,00	() [] -> .	von links nach rechts
14,00	! ~ ++ -- + - (TYP) * & sizeof	von rechts nach links
13,00	* / % (Rechenoperationen)	von links nach rechts
12,00	+ - (binär)	von links nach rechts
11,00	<< >>	von links nach rechts
10,00	< <= > >=	von links nach rechts
9,00	== !=	von links nach rechts
8,00	&	von links nach rechts
7,00	^	von links nach rechts
6,00		von links nach rechts
5,00	&&	von links nach rechts
4,00		von links nach rechts
3,00	?:	von rechts nach links
2,00	= += -= /= *= %= >>= <<= &= = ^=	von rechts nach links
1,00	, (Sequenz-Operator)	von links nach rechts

4.2. Der Preprocessor

Wie bereits erwähnt, fällt die erste Arbeit bei der C-Programmentwicklung, die der Compiler¹⁰ zu erledigen hat, an den C-Preprocessor. Er ist im wesentlichen für Textersetzungen und die Compilersteuerung¹¹ zuständig (sh. Seite 25).

4.2.1 Include-Direktive

Zum einen werden von diesem die `#include`-Zeilen ausgewertet, die angesprochenen Dateien (Include-Files) zur Compilationszeit eingebunden. Hierbei handelt es sich in der Regel um Headerfiles, d.h. um Quelltextdateien, in denen (nur) Deklarationen stehen. Solche Dateien tragen die Endung `.h`; es können aber prinzipiell auch andere Quelltextteile ausgelagert und *included* werden.

Hinweis: Wird die Datei in spitzen Klammern angegeben (`#include <stdio.h>`), so wird im festgelegten Pfad (bei UNIX ist das in der Regel `/usr/include`) nach der Datei (`stdio.h`) gesucht; wird die Datei in doppelten Hochkommata angegeben (`#include "myprog.h"`), so wird nur im aktuellen Verzeichnis (bzw. in dem eventuell angegebenen relativen oder absoluten Pfad) gesucht.

4.2.2 Define-Direktive und Makros

Weiterhin leistet der Preprocessor Textersatzfunktionen. Eine solche Definition hat die Form

```
#define <name> <ersatztext>
```

und sorgt dafür, daß überall, wo im ursprünglichen Quelltext `<name>` vorkam, in der erweiterten Quelltextfassung `<ersatztext>` steht. Dies gilt jedoch nicht innerhalb von Zeichenketten! `<name>` kann dabei einer der üblichen Namen sein, per Konvention schreibt man diesen in Großbuchstaben; der `<ersatztext>` darf irgendeine Zeichenkette sein, die sogar nötigenfalls über mehrere Zeilen gehen kann: in diesem Fall muß auf der vorherigen Zeile mit einem Backslash `\` abgeschlossen und in der ersten Spalte der Folgezeile fortgesetzt werden.

Beispiel:

```
#include <stdio.h>
#include "myprog.h"
#define MAXIMUM 120
#define MINIMUM 100
#define ANZAHL (MAXIMUM-MINIMUM) /* funktioniert auch! */
```

Darüber hinaus können aber auch über den Preprocessor Makros mit Parametern definiert werden.

Beispiel:

```
#define SQUARE(x) ((x)*(x))
```

¹⁰Häufig wird, wie hier, der Begriff *Compiler* für das Gesamtpaket bestehend aus C-Präprozessor, eigentlichem C-Compiler und ggf. noch dem Lader oder Linker verwendet.

¹¹Das Stichwort hierfür lautet *Bedingte Compilierung*.

Hiermit wird vereinbart, daß `SQUARE(x)` ein Makro ist, bei dem `x` dynamisch ersetzt wird. Eine Anweisung der Form

```
y = SQUARE(3);
```

wird vom Preprocessor somit expandiert zu

```
y = ((3)*(3));
```

Die Klammerung im Textersatz in der Definition von `SQUARE` ist übrigens nicht akademisch! Wird die Anweisung

```
y = SQUARE(x1+x2);
```

vom Preprocessor gelesen, so wird daraus bei obiger Definition korrekt die Zeile

```
y = ((x1+x2)*(x1+x2));
```

Betrachten wir dagegen folgende Definition:

```
#define SQUARE(x)  x*x
```

Die Anweisung

```
y = SQUARE(x1+x2)+x3;
```

wird damit (nur auf den ersten Blick überraschend) ersetzt zu

```
y = x1+x2*x1+x2+x3;
```

Im Gegensatz zu Funktionen ist es den Makros übrigens egal, welche Datentypen da auf sie niederprasseln: der Preprocessor macht schließlich nur eine einfache Textersetzung und keine semantische Typüberprüfung!

4.2.3 Trigraphen

Beim Programmieren arbeiten wir, bewußt oder unbewußt, stets mit mehreren Zeichensätzen gleichzeitig. Zum einen ist der ganze Code (bei uns in der Regel der 8-Bit-ASCII-Zeichensatz) des Betriebssystems und mehr oder weniger der Tastatur verfügbar, zum zweiten ist da der Zeichensatz, den die jeweilige Programmiersprache versteht.

Da nicht auf allen (vor allem älteren) Tastaturen jedes benötigte Zeichen für C zu finden ist, gibt es die sogenannten Dreizeichenfolgen (*Trigraphen*, *trigraph sequences*). Hierbei handelt es sich um Ersatzzeichenfolgen für ein bestimmtes Zeichen, wie sie in der nachstehenden Tabelle aufgeführt sind. So ist `a??(1??)` ein gültiger, wenn auch schwer lesbarer Ersatz für `a[1]`.

Soll etwas, z.B. eine Sequenz von mehreren Zeichen, nicht interpretiert werden, so kann stets mit dem Fluchtzeichen (Quotierungszeichen) `\` gearbeitet werden: die Anweisung

```
printf("Was ist das?\?!");
```

führt nach der Phase der Textersetzung durch den Preprocessor zur Anweisung

```
printf("Was ist das??!");
```

und damit zur Ausgabe

```
Was ist das??!
```

auf dem Bildschirm.

Dreizeichenfolge (Trigraph)	...ersetzt das Zeichen
<code>??=</code>	<code>#</code>

??([
??)]
??/	\
??'	^
??<	{
??>	}
??!	
??-	~

4.2.4 Bedingte Compilation

Schließlich sei noch auf eine weitere, in der Praxis sehr wichtige Anwendung von #define hingewiesen: die bedingte Compilation. Unter Bedingter Compilation versteht man die Möglichkeit, ein Quelltextstück nur unter einer gewissen Voraussetzung überhaupt compilieren zu lassen. Diese Voraussetzung ist das Definiertsein einer symbolischen Konstanten oder die Gleichheit mit einem bestimmten Wert. Folgendes Beispiel soll dies verdeutlichen; dabei werden gleichzeitig die Preprocessor-Direktiven #if, #ifdef, #ifndef, #else, #elif und #endif vorgestellt.

Beispiel:

```
#define TESTPHASE 1          /* während der Programmentwicklung */
                             /* wird TESTPHASE definiert als 1 */

#if TESTPHASE == 1
#   define PROGRAMMVERSION "Kolibri 1.0 [Testversion]"
#elif TESTPHASE == 2
#   define PROGRAMMVERSION "Kolibri 1.0 [Alpha-Release]"
#else
#   define PROGRAMMVERSION "Kolibri 1.0 [Final-Release]"
#endif
/* ..... */
printf("%s\n",PROGRAMMVERSION);
#ifdef TESTPHASE             /* ist TESTPHASE definiert worden? */
printf("Wir befinden uns in der Testphase des Programms.\n");
#endif
/* ..... */
#ifndef TESTPHASE           /* wenn nicht definiert, dann... */
printf("Wir befinden uns in der Abschlußphase...\n");
#endif
/* ..... */
```

4.3. Funktionen

Eine Funktion kennen Sie bereits: `main()`, das Hauptprogramm von C. In gleicher Weise können beliebig viele Funktionen¹² für ein C-Programm deklariert und definiert werden.

Wir haben aber gelegentlich auch schon eine weitere Funktion verwendet: `printf()`, eine Bibliotheksfunktion (vgl. den Abschnitt zu Bibliotheken und Headerfiles, Seite 29), die in `<stdio.h>` deklariert wird.

4.3.1 Formaler Aufbau einer Funktion

Der formale Aufbau einer Funktion ist recht einfach:

```
<ergebnistyp> <funktionsname> ( <parameterliste> )
{
  <deklarationen>
  <anweisungen>
}
```

Hierbei ist `<ergebnistyp>` irgendein vordefinierter oder selbstdefinierter Datentyp, die `<parameterliste>` eine – eventuell leere – komma-getrennte Aufzählung von Übergabeparametern.

Die Aufrufbarkeit und Gültigkeit wird auf Seite 43 im Kapitel zur Modularität eingehender besprochen. An dieser Stelle sei lediglich ausgeführt, daß eine Funktion eine jede andere (und sich selbst - Rekursion!) aufrufen kann, die dem Compiler zu diesem Zeitpunkt bereits bekanntgemacht worden ist. Durch das sogenannte Prototyping, i.e. das Voranstellen der Deklarationen der Funktionen vor die eigentlichen Implementationen (Definitionen) wird in der Praxis erreicht, daß jede Funktion jede andere aufrufen kann.

Ein ganz wichtiger, für Pascal-Programmierer ernüchternder Punkt: ANSI C kennt nur Wertübergaben, call by value! Eine Referenzübergabe (call by reference) in dem Sinne gibt es nicht! Wir werden weiter unten sehen, wie das Leben in C trotzdem weitergehen kann¹³.

Sehen wir uns einige einfache Beispiele an.

Beispiel:

¹²Genau genommen sind es laut Standard maximal 512 Funktionen pro Quelltextdatei; dieser Wert ist aber in der Praxis akademisch, da größere Softwareprojekte stets auf zahlreiche Dateien aufgeteilt werden. Sh. hierzu auch den Abschnitt zu Makefiles im Anhang dieses Scripts.

¹³Eine der (nicht objektorientierten) Erweiterungen von C++ gegenüber C ist auch die Einführung von Referenzen und Referenzparametern.

```

#include <stdio.h>

/* Prototypen: Bekanntmachen aller Funktionen, damit u.a.          */
   main() diese aufrufen kann.                                     */
float KreisFlaeche(float);

void main(void)
{
    float radius;
    printf("\nBitte einen Radius eingeben: ");
    scanf("%f",&radius);          /* Einlesen eines float-Wertes */
    printf("Radius: %f   Kreisfläche: %f",
           radius,KreisFlaeche(radius));
} /* end main */

float KreisFlaeche(float r)
{
    #define PI (3.1415926) /* #define kann überall im Source stehen */
    return PI*r*r;
} /* end KreisFlaeche */

```

Bei diesem kleinen Beispielprogramm ist einiges neu.

- a) Zunächst einmal sehen wir (entsprechend kommentiert) einen sogenannten Prototypen der Funktion KreisFlaeche(). Hier wird dem Compiler mitgeteilt, daß es eine solche Funktion geben und welchen Rückgabewert/typ sowie welche Parameterliste sie haben wird. Der Prototyp wird mit Semikolon abgeschlossen.
- b) Im Hauptprogramm main() ist der Aufruf von scanf() neu; dies ist das Gegenstück zu read(ln) bei Pascal. Wie printf() ist auch scanf() in <stdio.h> vereinbart; auf einige der zahlreichen Möglichkeiten von scanf() soll an anderer Stelle (sh. Seite 35) eingegangen werden. Hier nur die "lokale" Erläuterung: scanf() erwartet als ersten Parameter einen sogenannten Formatstring. In unserem Beispiel wird mit "%f" nur gesagt, daß ein float-Wert eingelesen werden soll. Als zweiter Parameter wird mit "&radius" gesagt, daß scanf() die Adresse der Variablen radius verwenden soll, um dort hinein den von Tastatur eingelesenen Wert abzuspeichern. Dies ist der angekündigte "Trick", wie trotz des call by value die Variable radius doch in der Funktion verändert werden kann: es wird einfach die Speicheradresse von radius ("by value") übergeben, damit kann de facto dann doch der Speicherplatz radius des Hauptprogramms angesprochen und verändert werden.
- c) Neu ist bei printf() eine entsprechende Erweiterung: auch hier ist nun der erste Parameter ("Radius: %f Kreisfläche: %f") ein sogenannter Formatstring, der zweite und dritte Parameter sind die float-Variable radius und der Funktionsaufruf KreisFlaeche(radius), der einen float-Wert zurückliefert. "%f" steht also auch hier wieder für die sachgemäße Interpretation der beiden Werte durch printf().
- d) Nach main() folgt nun (erstmal) eine weitere Funktion, hier KreisFlaeche(). Der formale Aufbau ist mit dem von main() vollkommen vergleichbar. Als Parameter wird ein float namens r vereinbart, als Rückgabetyt wird ebenfalls float benannt. Im Block der Funktion, d.h. zwischen den geschweiften Klammern, wird zunächst über den Preprocessor PI auf den allseits bekannten Wert 3.1415926 gesetzt, dann wird mit dem Schlüsselwort return

der Rückgabewert festgelegt. An dieser Stelle wird übrigens die Funktion auch schon wieder verlassen, selbst wenn anschließend noch weitere Anweisungen folgen sollten!

4.3.2 Parameter und Rückgabewerte

Die Parameter bei einer C-Funktion können von beliebigen Datentypen sein. Es gibt auch die Möglichkeit, auf die hier allerdings nicht näher eingegangen werden soll, Parameterlisten offen zu gestalten, d.h. nicht schon bei der Deklaration der Funktion festzulegen, wieviele Parameter die Funktion beim konkreten Aufruf haben soll.

Auf einen historischen Aspekt soll an dieser Stelle eingegangen werden: man unterscheidet heutzutage zwischen dem üblichen ANSI C, das auch hier in wesentlichen Teilen besprochen wird, und dem klassischen Kernighan-Ritchie-C (K&R-C), dem *old fashioned style*. Die unterschiedliche Definition von Funktionen ist eine der wesentlichen Änderungen von Kernighan-Ritchie-C zu ANSI C. Im K&R-C würde die obige Funktion KreisFlaeche() wie folgt deklariert und definiert werden.

```
float Kreisflaeche(radius)
float radius;
{
    #define PI (3.1415926)
    return PI*r*r;
} /* end of KreisFlaeche K&R-Style */
```

4.3.3 Bibliotheken und Headerfiles (Übersicht)

Wie erwähnt umfaßt der eigentliche Kern von C noch nicht einmal Routinen zur Terminal-Ein- und Ausgabe! (Hierzu speziell mehr im nachfolgenden Kapitel auf Seite 30.) Für fast alle weitergehenden Aufgabenstellungen muß C daher auf die Standardbibliothek (*standard library*) zugreifen.

Darunter versteht man eine Menge von Deklarationen und Funktionen, die als Bibliothek (bei UNIX: Dateien mit den Endungen¹⁴ .a (archive) oder .sl (shared library)) in Form von Object code mit eingebunden wird; die zugehörigen Deklarationen und Prototypen finden sich in den bereits mehrfach erwähnten Headerfiles, die mit der #include-Direktive durch den Preprocessor eingebunden werden.

In der folgenden Übersicht sind die Standard-Headerfiles gemäß ANSI aufgeführt, die bei einem Standard-UNIX-System üblicherweise in dem Verzeichnis /usr/include zu finden sind.

assert.h	Programmdiagnose
ctype.h	Zeichenklassen
errno.h	beinhaltet Fehlernummern
float.h	enthält Fließkomma-Grenzwerte
limits.h	enthält Ganzzahl-Grenzwerte
locale.h	Lokalisierung (Anpassung an spezielles System)
math.h	Mathematische Deklarationen und Routinen
setjmp.h	Globale Sprünge
signal.h	Signalverarbeitung
stdarg.h	Arbeiten mit variablen Argumentlisten
stddef.h	Deklaration allgemeiner Werte (z.B. von NULL)
stdio.h	Standard Ein-/Ausgabe (standard i/o)
stdlib.h	Hilfsfunktionen
string.h	Zeichenkettenverarbeitung
time.h	Datum und Uhrzeit

¹⁴Diese Dateinamenendungen sind spezifisch für das jeweilige UNIX-System. Die hier genannten sind beispielhaft für ein Hewlett-Packard HP-UX System (Version 9.0).

5 Terminal-Ein-/Ausgabe und Zeichenketten

In diesem Kapitel soll auszugsweise auf die Routinen der Standardbibliothek zur Ein- und Ausgabe eingegangen werden. Da ANSI diese Routinen in ihrer Wirkungs- und Anwendungsweise vorgeschrieben hat, können Programme, die sich nur dieser Funktionen bedienen, leicht von einem System auf ein anderes portiert werden.

In der Headerdatei `stdio.h` finden sich u.a. die folgenden Prototypen. Hier stehen auch die Deklarationen für die Standarddateien `stdin`, `stdout` und `stderr` (Fehlerausgabekanal).

```
/** ... auszugsweise ... */
extern int printf(const char *,...);
extern int scanf(const char *,...);
extern int sprintf(char *, const char *,...);
extern int sscanf(const char *, const char *,...);
extern int getchar(void);
extern char *gets(char *);
extern int putchar(int);
extern int puts(const char *);
```

5.1. Zeichenweise Ein- und Ausgabe

Der einfachste Mechanismus besteht darin, ein einzelnes Zeichen ein- bzw. auszugeben. Zur Eingabe eines Zeichens von Tastatur dient die Funktion `getchar()`. Prototyp:

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

`getchar()` liefert bei jedem Aufruf das nächste Zeichen im Eingabestrom (in der Regel `stdin`) oder den (ebenfalls in `stdio.h` definierten) Wert EOF (end of file) zur Kennzeichnung, daß der Eingabestrom geschlossen worden ist¹⁵.

Beispiel: Das nachfolgende Programm `readchar.c` liest ein Zeichen von Tastatur ein und gibt es zusammen mit seiner Nummer im ASCII-Code wieder aus. Beachten Sie jedoch: die Eingabe ist gepuffert, d.h. es muß erst [Return] gedrückt werden, bevor die Zuweisung an `zeichen` und die Ausgabe via `printf()` geschehen können¹⁶!

```
#include <stdio.h>
void main(void)
{
    int zeichen;          /* Beachten Sie: zeichen ist int! */
    zeichen=getchar();
    printf("Das Zeichen ist %c [ASCII-Nr.: %d]!\n",zeichen,zeichen);
} /* end main */
```

¹⁵EOF besitzt zwar üblicherweise den Wert -1, anständige C-Programme sollten jedoch besser die Konstante EOF verwenden um nicht von dieser Implementierung abhängig zu sein!

¹⁶Die `printf()`-Formatierungen `"%c"` und `"%d"` in dem Beispiel bewirken, daß die Variable `zeichen` einmal als char, ein zweites Mal als integer (digit) ausgegeben wird.

Neben dem bereits erwähnten `printf()` dient die Funktion `putchar()` zur Ausgabe eines einzelnen Zeichens auf den Ausgabestrom, in der Regel die Datei `stdout`. Prototyp:

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

Die Funktion `putchar()` gibt das übergebene Zeichen auf den Ausgabestrom aus; gleichzeitig liefert sie das ausgegebene Zeichen oder EOF zurück - EOF dann, wenn ein Fehler aufgetreten ist.

Hinweis: Bei den meisten Compilern sind `getchar()` und `putchar()` keine Funktionen, sondern als Makros realisiert. Auf diesen zunächst nicht so relevanten Unterschied soll an dieser Stelle nicht weiter eingegangen werden.

Beispiel: Das zugegebenermaßen nicht besonders aufregende nachstehende Programm liest ein Zeichen über `getchar()` von Tastatur ein und gibt es mittels `putchar()` wieder aus.

```
#include <stdio.h>
void main(void)
{
    int c = getchar();
    putchar(c);
} /* end main */
```

In der Datei `ctype.h` stehen u.a. die folgenden Prototypen. Die `isirgendwas()`-Routinen prüfen, ob ein gewisser Sachverhalt vorliegt, `isalpha()` prüft beispielsweise, ob das übergebene Zeichen ein (amerikanischer) Buchstabe ist, `isdigit()` ob es ein Ziffernzeichen ist, `islower()` ob es ein Kleinbuchstabe ist usw. Die Funktionen `tolower()` und `toupper()` wandeln (amerikanische) Buchstaben um in Klein- bzw. Großschreibung¹⁷.

```
/** ... auszugsweise ... */
extern int isalnum(int);
extern int isalpha(int);
extern int isdigit(int);
extern int islower(int);
extern int isprint(int);
extern int ispunct(int);
extern int isspace(int);
extern int isupper(int);
extern int tolower(int);
extern int toupper(int);
```

5.2. Zeichenketten (Strings)

Für weitergehende Ein- und Ausgabe (z.B. mittels `scanf()` und `printf()`, was beispielhaft bereits weiter vorne vorgestellt worden ist) sind Zeichenketten (Strings) erforderlich.

Zeichenketten können entweder statisch (wie `packed array of char` bei Pascal) oder dynamisch mit Pointern (wie der Datentyp `String` bei vielen Pascal-Dialekten) vereinbart werden. Sowohl auf die Array-, als auch die Pointer-Variante wird später noch ausführlicher eingegangen werden. Im Moment sollen uns die grundlegenden Deklarationen zur Anwendung in Zusammenhang mit `scanf()` und `printf()` genügen.

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

¹⁷Amerikanisch soll hierbei bedeuten, daß keine Umlaute, 'é', 'ß' und dergleichen berücksichtigt werden.

```

#include <string.h>
void main(void)
{
    char Statisch[20];
    char *Dynamisch = "Hello, world!";
    /** ... **/
    strcpy(Statisch,Dynamisch);
    printf("%s",Statisch);
    printf("%s",Dynamisch);
    putchar(Statisch[0]);
    /** ... **/
} /* end main */

```

Im obigen Beispielpogramm(auszug) strings.c wird ein statisches Array von 20 char-Speicherplätzen angelegt, die (für Pascal-Programmierer gewöhnungsbedürftig) mit 0 bis 19 adressiert werden können.

Wichtig: In C werden Zeichenketten mit einer terminierenden '\0' (ASCII-Zeichen Nr. 0) abgespeichert, die natürlich auch 1 Byte benötigt; daher "paßt" in die oben deklarierte Variable Statisch nur eine (sichtbare) Zeichenkette von maximal 19 Zeichen, wenn man nicht blaue Wunder erleben will!

Darunter wird die Variable Dynamisch durch "char *" vereinbart als Zeiger auf char (Pointer auf char). Dadurch erhält diese Variable zunächst nur den Speicherplatz, den ein Zeiger benötigt! Hier wird jedoch sofort – das geht in C generell – ein konstanter Text zugewiesen, so daß Dynamisch sofort über 14 Bytes verfügen kann. Mit der Funktion strcpy(), die in dem Headerfile string.h mit dem Prototypen char *strcpy(char *, const char *) vereinbart wurde, wird der Inhalt des zweiten Parameters in die Zeichenkette, die im ersten Parameter übergeben wird, kopiert. Der konkrete Funktionsaufruf strcpy(Statisch,Dynamisch); kopiert also den Inhalt von Dynamisch in die Variable Statisch - inklusive der terminierenden '\0'.

Unabhängig von der Deklarationsform kann bei beiden Variablen in gewohnter Weise auf die einzelnen Zeichen zugegriffen werden: Dynamisch[0] ist ein char, das hier das 'H' beinhaltet, Statisch[13] hat nach der Kopieraktion das Zeichen '\0' (ASCII-0) bekommen.

```

Skizze:      +-----+
              |H|e|l|l|o|,| |w|o|r|l|d|!|\0|?|?|?|?|?|?|
              +-----+
              0 1 2 3 4 5 6 7 8 9 10   13           19

```

5.3. Ein-/Ausgabe-Formatierung

Die wesentlichen Standardroutinen in C zur Ein- und Ausgabe von Zeichen(ketten) und numerischen Werten sind printf() und scanf().

5.3.1 Die Funktion printf()

Die Ausgaberroutine printf() ist in stdio.h deklariert mit dem Prototyp

```
(extern) int printf(const char *,...);
```

die drei Punkte deuten eine variabel lange Parameterliste an. Der erste Parameter bei printf() ist der sogenannte Formatstring: das ist das Muster, wie die Ausgabe aussehen soll. printf() schreibt auf den Standardausgabestrom (stdout, in der Regel der Bildschirm).

Beispiele:

```
printf("Konstanter Text"); /* Formatstring=konstanter Text */
printf("Zahl: %d",i);      /* %d bewirkt, daß i als int    */
                           /* aufbereitet ausgegeben wird */
printf("%d %c %x",i,j,k); /* i wird als int, j als char, */
                           /* k hexadezimal aufbereitet... */
```

Der Formatstring enthält also zwei verschiedene Arten von Objekten: gewöhnliche Zeichen (wie das Wort "Zahl" im Beispiel b)), die direkt in die Ausgabe geschrieben werden, und Formatierungen, die jeweils die entsprechende Umwandlung und Aufbereitung des nächsten Arguments von printf() bewirken. Jede solche Umwandlungsangabe beginnt mit einem Prozentzeichen % und endet mit einem Umwandlungszeichen. Dazwischen kann, in dieser Reihenfolge, angegeben werden:

- a) ein Minuszeichen: damit wird das Argument linksbündig ausgegeben;
- b) eine positive, ganze Zahl, die eine minimale Feldbreite bestimmt; benötigt das Argument mehr Stellen als angegeben, so werden ihm diese auch gegeben, benötigt es weniger, so wird mit Blanks aufgefüllt;
- c) ein Punkt, der die Feldbreite von der Genauigkeit (precision) trennt;
- d) eine positive, ganze Zahl, die die maximale Anzahl von Zeichen festlegt, die von einer Zeichenkette ausgegeben werden sollen, oder die Anzahl Ziffern, die nach dem Dezimalpunkt bei einer Gleitkommazahl ausgegeben werden, oder die minimale Anzahl von Ziffern, die bei einem ganzzahligen Wert ausgegeben werden sollen;
- e) der Buchstabe h oder H, wenn short ausgegeben werden soll, oder der Buchstabe l oder L, wenn das Argument long ist.

Hinweis: Wenn das Zeichen nach % keines der obigen Zeichen und kein Umwandlungszeichen ist, dann ist die Wirkung von printf() undefiniert!

Nachstehend eine kurze (nicht vollständige) Übersicht über wichtige Formatierungszeichen.

Symbol	...steht für...
d	dezimale Ganzzahl, int
x	hexadezimale Ganzzahl, int
u	vorzeichenlose Ganzzahl, unsigned int
hd	kurze Ganzzahl, short int
ld	dezimale Ganzzahl, long int
f	Gleitkommazahl, float
lf	Gleitkommazahl, double
c	einzelnes Zeichen, char

Nachstehend ein kleines Beispiel zur Illustration: Zunächst das Programm, dann das Ablauflisting.

```
#include <stdio.h>
void main(void)
{
    char *txt="Eine kleine Textzeile";
    int i=123456;

    printf("\n:%s:",txt);
    printf("\n:%15s:",txt);
    printf("\n:%-10s:",txt);
    printf("\n:%15.10s:",txt);
    printf("\n:%-15.10s:",txt);
    printf("\n:%-10.5s:",txt);
    printf("\n:%.10s:",txt);
    printf("\n");

    printf("\n:%20d:",i);
    printf("\n:%-10d:",i);
    printf("\n:%5.3d:",i);
    printf("\n");

} /* end main */
```

Und das zugehörige Ablauflisting:

```
:Eine kleine Textzeile:  
:Eine kleine Textzeile:  
:Eine kleine Textzeile:  
:   Eine klein:  
:Eine klein      :  
:Eine           :  
:Eine klein:  
  
:               123456:  
:123456       :  
:123456:
```

5.3.2 Die Funktion scanf()

Die Funktion scanf() ist die zu printf() analoge Einleseroutine, die ebenfalls in stdio.h deklariert ist mit dem Prototypen

```
int scanf(const char *,...);
```

scanf() liest Zeichen aus dem Standardeingabestrom (stdin), wobei die Verarbeitungsweise wieder über einen Formatstring kontrolliert wird. scanf() hört auf, wenn die Format-Zeichenkette vollständig abgearbeitet ist, oder aber wenn ein Eingabefeld nicht zur Umwandlungsangabe paßt. Als Funktionsresultat wird die Anzahl erfolgreich erkannter und zugewiesener Eingabefelder zurückgeliefert. Am Eingabeende wird EOF zurückgeliefert. Der nächste Aufruf von scanf() beginnt dann seine Arbeit unmittelbar nach dem zuletzt umgewandelten Zeichen.

Übersicht: Elementare scanf()-Formatstrings (Auszug)

Zeichen	Argument	Eingabe
d, i	int*	dezimal, ganzzahlig, int
u	int*	dezimal ohne Vorzeichen, unsigned
c	char*	ein einzelnes Zeichen
s	char*	eine Zeichenkette (ohne Hochkommata), das Zeichen '\0' wird von C automatisch hinzugefügt. Wichtig: Es wird allerdings nur bis zum ersten Whitespace (Leerzeichen, Tabulator etc.) eingelesen!
e, f, g	float*	Gleitkommazahlen, float

Einige Beispiele für scanf()-Aufrufe:

```
int      i;
float    f;
char     txt[80];
char     c;
/* ... */
scanf("%d",&i);          /* Wichtig: Adreßoperator & vor i !!! */
scanf("%f",&f);          /* Denn C kennt nur call by value !!! */
scanf("%s",txt);        /* Arrays sind intern Adressen, daher */
                          /* muß hier kein Adreßoperator genom- */
                          /* men werden! */
scanf("%c",&c);          /* Adresse von txt[0] */
scanf("%c",&(txt[0]));
scanf("%u",&i);
```

Warnung: Noch einmal ausdrücklich: einer der häufigsten Anfängerfehler ist die Formulierung `scanf("%d",i);`

Hiermit wird nicht auf den Speicherplatz `i` eingelesen, sondern dorthin in den Hauptspeicher, wohin der momentane Wert von `i` (wenn überhaupt) gerade zeigt. Unter UNIX wird dieser Fehler normalerweise bemerkt, auf dem PC kann er beliebige Nebenwirkungen haben...

Aus Platzgründen soll es bei diesem kurzen Einblick bleiben; neben den hier vorgestellten beiden Grundroutinen sind in `stdio.h` natürlich noch eine ganze Reihe weiterer Routinen zum (formatierten) Einlesen aus Dateien oder Speicherbereichen vorhanden (`fprintf()` und `fscanf()`, `sprintf()` und `sscanf()`). Speziell `fprintf()` wird dann benötigt, wenn Ausgaben auf den Fehlerkanal `stderr` geleitet werden sollen, wie die folgende Skizze illustriert.

```
/* error1.c */
#include <stdio.h>
void main(void)
{
    printf("Diese Ausgabe geht auf stdout...\n");
    fprintf(stderr,"Diese Ausgabe geht auf stderr...\n");
} /* end main */
```

Ablauflisting:

1. Bei Aufruf "error1" (stdout und stderr gehen auf den Bildschirm)

Diese Ausgabe geht auf stdout...

Diese Ausgabe geht auf stderr...

2. Bei Aufruf "error1 > anyfile"

Diese Ausgabe geht auf stderr...

(Die erste Textzeile ist in der Datei anyfile zu finden!)

6 Kontrollstrukturen

Im folgenden soll kurz auf die Kontrollstrukturen in ANSI C eingegangen werden. Dabei wird auf Ihren soliden Pascal-Kenntnissen¹⁸ aufgesetzt, die Beispiele aus Platzgründen werden also bewußt sehr kurz gehalten. Ebenso wird davon ausgegangen, daß Sie die Spielregeln der strukturierten Programmierung bereits kennengelernt haben.

6.1. Einzelanweisungen und Blöcke

Jeder Ausdruck in C wird zu einer Anweisung, wenn ihm ein Semikolon folgt! Die folgenden Zeilen beschreiben also in diesem Sinne einzelne C-Anweisungen.

```
x=y=z=0;
i++;
printf("\f\v\tLook here, are we \bnormal?\n");
```

Mit geschweiften Klammern { und } wird ein Block festgelegt. Ein solcher Block ist syntaktisch eine Anweisung und darf infolgedessen überall dort stehen, wo der Syntax gemäß eine Anweisung plaziert werden darf.

Innerhalb eines Blockes können (zu Beginn) (lokale) Variablen deklariert werden! (Vgl. hierzu das nachfolgende Kapitel zur Modularität.) Nach der schließenden geschweiften Klammer steht hierbei kein Semikolon.

Beispiel:

```
void main(void)
{
    /* Hier beginnt der Block */
    int i;      /* i ist gültig nur innerhalb dieses Blockes */
    /* ... */
}              /* Hier endet der Block und damit die Gültigkeit von i */
```

6.2. Logische Ausdrücke und Verzweigung (if, if-else)

Mit der if-Anweisung bzw. if-else-Anweisung werden ein- oder zweifache Alternativen formuliert. Die Syntax unterscheidet sich etwas von der aus Pascal bekannten:

```
if ( <expression> ) <statement1> else <statement2>
```

Natürlich ist der else-Zweig optional. Die runden Klammern bei dem logischen Ausdruck sind erforderlich.

Beispiel:

```
if ( x != 0 )
    z /= x;
else
    z = -1;
```

¹⁸Ersatzweise genügt es auch, wenn Sie BASIC oder womöglich sogar COBOL kennen. Pascal wäre aber schon gut...

Zu beachten ist hierbei, daß "logisch" bei C stets numerisch (ganzzahlig) bedeutet! Das heißt, der im if-Konstrukt auftretende Ausdruck kann generell jeden beliebigen numerischen Wert annehmen: wie bereits erwähnt wird 0 als FALSE, jeder andere Wert als TRUE interpretiert.

Obiges Beispiel läßt sich also umschreiben zu:

```
if (x)
    z /= x;
else
    z = -1;
```

Warnung: Ein häufiger Fehler (vor allem bei Menschen aus der Pascal-Welt) ist die Formulierung

```
if (x=0) ...;
```

in C wird hier der Variablen x der Wert 0 zugewiesen, das Ergebnis des Ausdruckes ist damit 0 (=FALSE), und ggf. wird der else-Zweig abgearbeitet! Korrekt muß es also

```
if (x==0) ...;
```

lauten!

6.3. Iterationen (while, for, do-while)

C bietet die drei gewohnten Schleifen an: die kopfgesteuerte while-Schleife, die fußgesteuerte do-while-Schleife und eine Schleife mit dem Schlüsselwort for, die weit mehr als die aus Pascal bekannte Zählschleife beinhaltet.

6.3.1 Die while-Schleife

Die kopfgesteuerte while-Schleife hat die Syntax

```
while ( <expression> ) <statement>
```

und wird solange abgearbeitet, wie <expression> einen Wert ungleich 0 besitzt.

Beispiel:

```
while ( i < MAX )
{
    sum += i++; /* i wird nach der Summation inkrementiert! */
}
/* Die geschweifte Block-Klammerung wäre hier nicht zwingend erforderlich. */
```

6.3.2 Die for-Schleife

Die for-Schleife in C ist ein sehr mächtiges Konstrukt, das durchaus nicht nur für die klassischen Zählschleifen verwendet wird.

Die Syntax hat die Form

```
for ( <expression1> ; <expression2> ; <expression3> ) <statement>
```

Mit Ausnahme der später zu besprechenden continue-Anweisung ist die for-Schleife äquivalent zu

```
<expression1>;
```

```
while (<expression2>)
{
    <statement>
    <expression3>;
}
```

Das bedeutet: <expression1> ist eine Anweisung, die vor dem eigentlichen Schleifenbeginn ausgeführt wird. <expression2> ist die logische Bedingung, die die weitere Schleifenverarbeitung regelt: solange <expression2> erfüllt (d.h. ungleich 0) ist, wird <statement> ausgeführt. Schließlich ist <expression3> eine Anweisung, die zum Abschluß eines jeden Schleifendurchgangs ausgeführt wird, die sogenannte Reinitialisierung.

Beispiel: Die nachstehende for-Anweisung summiert alle ganzen Zahlen von 0 bis 9 in der Variablen sum auf; dabei wird sum kompakterweise auch noch im Kopf der for-Schleife initialisiert.

```
for (sum=i=0; i<10; i++)
{
    sum += i;
}
```

Beispiel: Soll bei for nichts initialisiert werden, so kann ein for-Konstrukt auch so aussehen:

```
for ( ; i<10; i++ )
{
    sum += i;
}
```

Beispiel: Gelegentlich werden Endlosschleifen benötigt, z.B. wenn innerhalb einer Funktion mit return schon vorzeitig aus einer Schleife herausgesprungen wird. Das kann dann so aussehen:

```
for (;;)
{
    /* ... */
    if (x==0)
        return -1;
    /* ... */
} /* end for */
```

6.3.3 Die do-while-Schleife

In der Praxis etwas seltener wird die fußgesteuerte do-while-Schleife eingesetzt. Die Syntax ist

```
do { <statements> } while ( <expression> );
```

dabei werden die <statements> solange abgearbeitet, wie <expression> einen Wert ungleich 0 besitzt, also TRUE ist. Beachten Sie bitte, daß dies die negierte Formulierung zum repeat-until-Konstrukt in Pascal ist!

Beispiel:

```
do
{
    sum += i;
    i++;
} while ( i < MAX );
```

Dieses Beispiel würde in Pascal so aussehen:

```
repeat
    sum := sum + i;
    i := i + 1
until i >= MAX;
```

6.4. Mehrfachverzweigung (switch) und Marken

C kennt auch die Mehrfachverzweigung - das case von Pascal ist hier die switch-Anweisung. Die allgemeine Syntax hat die Form

```
switch ( <expression> )
{
    case <const-expr> : <statements>
    case <const-expr> : <statements>
    default           : <statements>
}
```

Hierbei wird <expression> mit den einzelnen konstanten (und paarweise verschiedenen) Ausdrücken hinter den Schlüsselwörtern case verglichen; ist irgendwo Gleichheit festzustellen, so wird dort eingesprungen, und die dahinter stehenden Anweisungen werden ausgeführt. Dabei werden dann sämtliche weiteren Anweisungen innerhalb des switch-Konstruktes abgearbeitet, also auch die, die hinter einer späteren switch-Marke stehen!

Um dies zu verhindern, kann mit break (vgl. den nächsten Abschnitt) der Ausstieg aus dem Konstrukt befohlen werden, wie im entsprechenden Beispiel dort gezeigt werden wird.

Trifft keine der case-Marken zu, so wird bei dem Schlüsselwort default eingesprungen; diese Marke darf jedoch auch fehlen, in diesem Fall findet bei switch dann keine Verarbeitung statt¹⁹.

6.5. Abbruchmöglichkeiten (continue, break, return, exit)

Es ist manchmal sinnvoll, einen strukturierten Ablauf vorzeitig abubrechen. Dies kann um den Preis umfangreicherer und schwerer zu lesenden Codes stets durch die Einführung logischer Flags geschehen, die den weiteren Ablauf z.B. eines Teiles der Anweisungen einer Schleife regeln. C stellt aber einige Möglichkeiten bereit, gezielt einen solchen vorzeitigen Ausstieg (und nur diesen) vorzunehmen.

Die Anweisung

¹⁹Insbesondere stürzt das C-Programm im Gegensatz zu der Erfahrung beim case-Konstrukt von Standard-Pascal auf keinen Fall ab!

```
continue;
```

bewirkt innerhalb einer Schleife, daß der restliche Schleifenkörper übersprungen und es mit ggf. dem nächsten Schleifendurchlauf weitergeht. Bei verschachtelten Schleifen bezieht sich continue naheliegenderweise auf die innerste Ebene.

Beispiel:

```
for (i=0; i<n; i++)
{
    /* ... */
    if (func(i)<0)
        continue;                /* negative Werte werden übersprungen */
} /* end for */
```

Die Anweisung

```
break;
```

ist einfach: stößt das Programm innerhalb einer Schleife oder switch-Anweisung auf ein break, so wird die entsprechende Struktur (vorzeitig) verlassen.

Beispiel:

```
switch ( i )
{
    case 1 : printf("\nEins\n");
             break;
    case 2 : printf("\nZwei\n"); /* ohne break; wird im Falle von i==2 */
             /* in der nächsten Zeile weitergemacht */
    case 3 : printf("\nDrei\n");
             break;
    default: printf("\nKeine der Zahlen...\n");
             break;           /* nicht erforderlich, aber guter Stil */
} /* end switch /
```

Die return-Anweisung wurde bereits in Zusammenhang mit Funktionen vorgestellt: stößt die Abarbeitung auf ein return (mit oder ohne einen darauffolgenden Rückgabewert), so wird die betreffende Funktion beendet. Geschieht dies innerhalb von main(), so endet das gesamte Programm.

Mit

```
return 5;
```

oder

```
return(5);
```

wird der Wert 5 dabei zurückgeliefert.

C stellt einen weiteren Mechanismus (sozusagen als Notausstieg) bereit: mit der Standardfunktion exit() wird das (gesamte) Programm beendet, gleichgültig, aus welcher Funktion heraus exit() aufgerufen wird. Der Prototyp in stdlib.h ist

```
void exit(int status);
```

Als status kann dabei ein Wert an die aufrufende Instanz (Betriebssystem) zurückgeliefert werden. Vordefiniert sind die beiden symbolischen Konstanten EXIT_SUCCESS und

EXIT_FAILURE, die für einen inhaltlich erfolgreichen bzw. fehlerbehafteten Programmausstieg stehen.

In einem korrekten Zweig sollte also mit

```
exit(EXIT_SUCCESS);
```

deutlich gemacht werden, daß das Programm ohne Fehler abgearbeitet worden ist; dementsprechend ist

```
exit(EXIT_FAILURE);
```

das Signal dafür, daß im Programm ein (nicht behebbarer) Fehler aufgetreten ist.

6.6. Sprünge (goto)

Neben den bisher vorgestellten Ablaufstrukturen gibt es auch in C ein goto, das aber seit der Erfindung der strukturierten Programmierung nicht mehr verwendet wird.

7 Modularität, Gültigkeitsbereiche und Speicherklassen

C ist eine Programmiersprache, die modulares Arbeiten unterstützt. Als Modul soll im folgenden der Quelltext einer einzelnen Datei, eines einzelnen Sourcefiles, angesehen werden. Das Programm seinerseits kann aus 1 bis endlich vielen Modulen bestehen, jedes Modul aus 0 bis endlich vielen²⁰ Funktionen.

Beispiel: Ist der gesamte Quelltext eines Programms verteilt auf die Dateien main.c, sub1.c und sub2.c, so besitzt dieses Programm drei Module. Unter UNIX können diese nun mit dem Compileraufruf

```
cc main.c sub1.c sub2.c -o myprog
```

compiliert und zu einem ausführbaren Programmfile namens myprog zusammengebunden werden.

7.1. Modularität und Gültigkeitsbereiche

C kennt zwar keine Verschachtelung von Funktionen (wie Pascal); da aber in jedem Block (nicht nur in dem äußersten) Variablen deklariert werden können, ist hier zumindest in dieser Hinsicht eine entsprechende Blockstruktur wie bei Pascal zu finden. Wird in einem inneren Block eine Variable *i* deklariert, so überdeckt diese für die Dauer dieses Blockes eine eventuell in einem äußeren Block oder auf Grundebene (außerhalb aller Blöcke) deklarierte Variable (oder Konstante oder Funktion) *i*.

Ein C-Programm besteht aus einer Reihe von externen (globalen) Objekten, das können Variablen oder Funktionen sein. (main() ist auf jeden Fall ein solches externes Objekt.) Dabei wird extern als Kontrast zu intern verwendet und bezeichnet alle Objekte, die außerhalb einer Funktion vereinbart werden. Die Variablendeklarationen innerhalb einer Funktion führen dementsprechend zu internen Variablen. Auch die Variablen(namen) in den Parameterlisten sind in diesem Sinne interne Größen. Funktionen sind stets extern.

Per Default haben externe Objekte die Eigenschaft, daß alle Verweise auf sie mit gleichem Namen auch das gleiche Objekt bezeichnen, sogar aus Funktionen heraus, die separat compiliert worden sind. Dies nennt man externe Bindung²¹.

Gültig ist eine interne Variable stets nur in der Einheit, in der sie deklariert wurde; eine externe Variable ist im gesamten Programm gültig – mit der Einschränkung des Überdecktwerdens durch gleichnamige lokale Variablen.

Funktionen, die in C immer auf der globalen Ebene stehen müssen, sind von sich aus extern, d.h. aus jedem Modul kann auf sie zugegriffen werden, Prototyping vorausgesetzt. Dies kann durch explizites Hinzufügen des Schlüsselwortes extern vor den Rückgabetyt betont werden.

Beispiel:

```
extern float power(float,int);
```

²⁰Bei ANSI-C ist diese Anzahl auf 255 limitiert.

²¹Dieses Verhalten ist das der globalen Variablen in Pascal.

Dagegen kann die Gültigkeit und Aufrufbarkeit einer Funktion auf das betreffende Modul beschränkt werden, indem vor den Rückgabebetyp das Schlüsselwort `static` gesetzt wird.

Beispiel:

```
static float power(float,int);
```

Nun ist `power()` nur noch von Funktionen desselben Moduls aufrufbar!

7.2. Speicherklassen (`auto`, `static`, `register`, `extern`)

Es gibt grundsätzlich zwei Speicherklassen in C: automatisch (`auto`) und statisch (`static`). Zusammen mit dem Kontext der Deklaration eines Objektes (z.B. einer Variablen) bestimmen verschiedene Schlüsselwörter die zu verwendende Speicherklasse.

Automatische Objekte existieren (nur) lokal in einem Block und werden bei Verlassen des Blockes zerstört. Deklarationen innerhalb eines Blockes kreieren automatische Objekte, wenn keine Speicherklasse explizit angegeben wird. Mit dem Schlüsselwort `register` deklarierte Objekte sind automatisch, werden jedoch nach Möglichkeit in Hardware-Registern verwaltet.

Statische Objekte können lokal in einem Block, in einer Funktion oder auch außerhalb von allen Blöcken deklariert werden; sie behalten ihre Speicherplätze und Werte aber in jedem Fall bei Verlassen von und beim Wiedereintritt in Blöcke und Funktionen bei! In einem Block (und in einer Funktion) werden Objekte mit dem Schlüsselwort `static` als statisch deklariert. Außerhalb von allen Blöcken sind Objekte stets statisch. Mit `static` können sie lokal für ein Modul (Quelltextfile) vereinbart werden, dadurch erhalten sie eine sogenannte interne Bindung (*internal linkage*); für ein gesamtes Programm werden sie global bekannt, wenn keine Speicherklasse angegeben wird oder aber durch Verwendung des Schlüsselwortes `extern`, dadurch erhalten sie externe Bindung (*external linkage*).

Übersicht: Speicherklassen, Gültigkeitsbereiche und Lebensdauer

Klasse	Gültigkeit	Lebensdauer	Automatische Initialisierung?
<code>auto</code>	Block	Block	nein
<code>register</code>	Block	Block	nein
<code>extern</code>	Programm	Programmmlauf	ja
<code>static (blockintern)</code>	Block	Programmmlauf	ja
<code>static (außerhalb aller Blöcke)</code>	Quelldatei (Modul)	Programmmlauf	ja

7.3. Attribute für Datentypen: const und volatile

ANSI C kennt zwei Attribute für Datentypen: const und volatile. Diese Attribute können mit jeder Typangabe gekoppelt auftreten.

7.3.1 Das Schlüsselwort const

Ein Objekt mit dem Attribut const (für konstant) muß initialisiert werden und kann anschließend nicht mehr verändert werden, darf also insbesondere nicht neue Werte zugewiesen bekommen. Der Compiler hat die Möglichkeit, const-Objekte in anderen Speicherbereichen zu verwalten als normale Variablen.

Beispiel:

```
const double PI=3.1415926;
```

7.3.2 Das Schlüsselwort volatile

Mit dem Attribut volatile wird dem Compiler mitgeteilt, daß das entsprechende Objekt durch externe Einflüsse geändert werden kann, z.B. durch die Systemuhr. Der Compiler benötigt eine solche Angabe, damit er nicht anhand des Programmcodes davon ausgeht, daß sich ein Objekt nicht ändert und eventuell durch eine ansonsten sinnvolle Optimierung das Programm verfälscht.

8 Höhere Datentypen

C kennt mit Ausnahme von Mengen (*sets*) die strukturierten und dynamischen Datentypen wie Pascal auch: Arrays (Felder, Vektoren), Strukturen (feste und variante Records), Zeiger (Pointer) und Files (Dateien), die hier jedoch in einem eigenständigen Kapitel behandelt werden. Und wenn man Mengen doch benötigt, dann helfen vielleicht die Bit-Felder weiter, die in diesem Kapitel in einem eigenen Abschnitt eingeführt werden.

8.1. Eindimensionale Arrays

Eine über einen ganzzahligen Index ansprechbare endliche Sequenz von Speicherplätzen desselben Typs heißt auch in C Array oder Feld. Deklaration und Verwendung werden im folgenden Beispiel `arrays1.c` demonstriert.

Beispiel:

```
/* arrays1.c */
#include <stdio.h>
void main(void)
{
    int i, a[10], j ;
    char s[10]="Hello!";

    for (i=0; i<10; i++)
        a[i]=i*i;
    printf("\n&i=%d",&i);           /* Adresse von i ausgeben */
    printf("\n&j=%d",&j);           /* Adresse von j ausgeben */
    printf("\na=%d",a);             /* Was ist a, das Array? */
    printf("\n&a=%d",&a);          /* Adresse von a */
    printf("\n&a[0]=%d",&(a[0]));   /* Adresse von a[0] */
    printf("\n&a[1]=%d",&(a[1]));   /* Adresse von a[1] */
    printf("\na[0]=%d",a[0]);       /* Wert in a[0] */
    printf("\na[1]=%d\n",a[1]);     /* Wert in a[1] */
    for (i=0; i<10; i++)
        printf("s[%d]=%d ",i,s[i]); /* Was steht in s? */
    putchar('\n');
} /* end main */
```

Das Ablauflisting zu `arrays1.c`:

```
&i=2063807476
&j=2063807432
a=2063807436
&a=2063807436
&a[0]=2063807436
&a[1]=2063807440
a[0]=0
a[1]=1
s[0]=72 s[1]=101 s[2]=108 s[3]=108 s[4]=111 s[5]=33 s[6]=0 s[7]=0
s[8]=0 s[9]=0
```

Zu beachten ist hierbei:

- a) Der Name des Arrays (z.B. a in arrays1.c) steht bereits für die Adresse des Feldes; d.h. z.B. bei scanf() muß kein Adreßoperator & mehr angegeben werden!
- b) Der Indexbereich eines Arrays beginnt stets bei 0. Das oben deklarierte Array `int a[10];` besitzt somit zwar zehn Komponenten, aber `a[10]` gibt es nicht!

Bereits in Abschnitt 4.2. wurden Zeichenketten vorgestellt. Auch dies sind Arrays. So deklariert und definiert in dem obigen Beispielprogramm arrays1.c die Zeile

```
char s[10]="Hello!";
```

eine Zeichenkette s mit zehn Speicherplätzen, wovon jedoch auch das Stringendezeichen ASCII-0 einen Platz belegt! `s[0]` ist hier 'H' usw.

Noch einmal sei daran erinnert, daß in der Headerdatei string.h eine ganze Reihe von Funktionen für Strings deklariert sind. Sehen Sie sich unter UNIX doch einmal die Datei `/usr/include/string.h` bzw. bei einem PC-Compiler die Datei string.h im entsprechenden INCLUDE-Verzeichnis an!

Und noch etwas: auch wenn die Deklaration `char *s2;` etwas anderes als ein Array, nämlich einen Zeiger, beschreibt, so kann `s2` dennoch syntaktisch genauso wie das obige Array s verwendet werden: hier kann also ebenfalls mit `s2[0]` auf das erste Element von `s2` zugegriffen werden, sofern ein solches existiert!

8.2. Mehrdimensionale Arrays

Selbstverständlich können Arraystrukturen auch geschachtelt werden. Ein zweidimensionales Array von int-Werten kann z.B. deklariert werden durch

```
int Matrix[10][20];
```

Hiermit steht eine Datenstruktur mit 10*20 int-Speicherplätzen zur Verfügung; anschaulich kann auf die Elemente in den 10 Zeilen und 20 Spalten zugegriffen werden wie folgt.

```
Matrix[0][0]=1;           /* das allererste Element           */
Matrix[9][19]=200;       /* das allerletzte Element          */
Matrix[9][0]=181;       /* das erste Element der letzten Zeile */
```

Bemerkung: Wie zuvor ist auch hier (natürlich) der Name Matrix bereits die Adresse des Feldes!

8.3. Strukturen (struct)

Was Pascal die Records sind, heißt bei C struct (Struktur oder Verbund). Diese werden z.B. für die Arbeit mit Datenbanken benötigt; nachstehend sehen wir uns ein kleines Beispiel mit einem struct an. Spätestens hier wird im übrigen die Verwendung von typedef sehr sinnvoll!

```
/* structs1.c */
#include <stdio.h>
#define STRLEN 128

struct Personal
{
    char  Nachname[STRLEN];
    char  Vorname[STRLEN];
    int   PersonalNr;
    float Gehalt;
} personall;

typedef struct Einsatz
{
    char  Name[STRLEN];
    char  Fach[STRLEN];
    int   Stunden;
} EINSATZ;

void main(void)
{
    struct Personal personal2;
    EINSATZ einsatz;

    strcpy(personall.Nachname, "Müller");
    strcpy(personall.Vorname, "Alfons");
    personall.PersonalNr=111;
    personall.Gehalt=999.99;
    personal2=personall;

} /* end main */
```

In diesem Beispiel wird ein Datentyp "struct Personal" vereinbart mit den Komponenten Nachname, Vorname, PersonalNr und Gehalt der entsprechend genannten Datentypen. Damit ist noch kein Speicherplatz allokiert worden. Dies geschieht erst durch das Anfügen des Bezeichners "personall" hinter die Strukturdefinition! personall ist hier also eine (externe) global gültige Variable mit den genannten Komponenten. Der Zugriff, wie weiter unten in dem Beispielprogramm structs1.c zu sehen, geschieht (wie bei Pascal) durch den Punktoperator: personall.Nachname greift auf die Komponente Nachname der Variablen personall zu.

Wie im Hauptprogramm zu sehen ist, muß allerdings bei jeder neuen Deklaration einer Variablen von diesem Strukturtyp auch das Wort struct mitgeschrieben werden, was zumindest unbequem ist. Die Typendefinition EINSATZ im obigen Beispiel zeigt, wie es angenehmer geht: innerhalb der typedef-Klausel wird die Struktur Einsatz deklariert und dieser

dann der Synonymname EINSATZ (in Großbuchstaben) gegeben. Eine Variable einsatz kann dann wie oben im Hauptprogramm einfach durch

```
EINSATZ einsatz;
```

deklariert werden.

Natürlich können Strukturen (wie alle anderen höheren Datentypen) auch geschachtelt werden. Legen wir die Deklarationen aus dem obigen Beispiel structs1.c zugrunde, so kann mit

```
struct STRUKTUR
{
    struct Personal  p;
    EINSATZ          e;
} struktur;
```

eine Variable struktur vom Typ struct STRUKTUR vereinbart werden; korrekt sind dann die Zugriffe struktur.p.PersonalNr oder struktur.e.Stunden.

8.4. Variante Strukturen (union)

Während beim normalen struct alle Komponenten im Speicher hintereinander liegen, gestattet der Datentyp union das umzusetzen, was in Pascal variante Records sind: mehrere Komponenten liegen übereinander, so daß ein- und derselbe Platz für verschiedenartige Daten genutzt werden kann. Die Syntax ist ansonsten wie bei den (festen) structs.

Auch hier am besten ein kleines Beispiel.

```
/* unions1.c */

#include <stdio.h>
#include <string.h>

union U2
{
    unsigned char  s[10];
    unsigned int   i;
} u2;

void main(void)
{
    printf("sizeof(struct U2): %d\n",sizeof(union U2)); /* ==10 */
    printf("Sizeof(struct u2): %d\n",sizeof(u2));

    u2.i=123;
    strcpy(u2.s,"A");
    /* Nun ist (auf dem PC!) u2.i==65 (ASCII-Code) */
} /* end main */
```

Verwendet werden solche varianten Strukturen (unions) z.B. dann, wenn in einem Datenbestand sich ausschließende verschiedenartige Ausprägungen auftreten können und man die jeweils auf gleichviel Speicherplatz unterbringen möchte.

Beispiel: Im PC-Bereich werden unions beispielsweise eingesetzt für die Arbeit mit den Registern, die einmal byteweise, ein anderes Mal in Worteinheiten angesprochen werden müssen²². Der Turbo C Compiler von Borland deklariert²³ deshalb die folgenden Strukturen und die Union REGS:

```
struct WORDREGS                                /* wortweise adressierte Register */
{
    unsigned int    ax, bx, cx, dx, si, di, cflag, flags;
};

struct BYTEREGS                                /* byteweise adressierte Halbreister */
{
    unsigned char   al, ah, bl, bh, cl, ch, dl, dh;
};

union    REGS                                  /* die variable Verbindung dieser beiden */
{
    struct WORDREGS x;
    struct BYTEREGS h;
};
```

Wird nun eine Variable

```
union REGS reg;
```

deklariert, so kann mit

```
reg.x.ax = 0xFF00;
```

```
    /* höherwertiges Byte auf 0xFF setzen, niedrigerwertiges Byte auf 0x00
```

```
*/
```

dafür gesorgt werden, daß reg.h.ah den Wert 0xFF, reg.h.al den Wert 0x00 erhält.

²²Für die Leser und Leserinnen mit Assembler-Kenntnissen: Das mit AX bezeichnete Register besitzt die beiden ein Byte großen Komponenten AH (*high byte*) und AL (*low byte*). Entsprechendes gilt für die anderen Register eines Intel-80x86-kompatiblen Prozessors.

²³Diese Deklarationen stehen in der DOS-spezifischen Headerdatei include\dos.h.

8.5. Bit-Felder

Ein Spezialfall einer Struktur sind die sogenannten Bit-Felder (*bit fields*)²⁴. Hier wird in einer Strukturvereinbarung durch einen Doppelpunkt getrennt jeweils vorgeschrieben, wieviele Bits eine Komponente umfassen soll. Dieser Ansatz eignet sich insbesondere gut zur Verwaltung von Flags, logischen Merkern.

Wichtig: Fast alle Aspekte von Bit-Feldern sind implementierungsabhängig; es gibt keine Arrays von Bit-Feldern, und Bit-Felder haben keine Adressen, so daß der Adreßoperator auf sie nicht angewendet werden kann.

Auch hier wieder ein Beispiel. (Mit den beiden #define-Direktiven werden die symbolischen Konstanten TRUE und FALSE auf 1 und 0 gesetzt, wobei allerdings dem konkreten Rechner überlassen bleibt, in welcher internen Darstellung und Speicherbreite 1 und 0 ermittelt werden, daher die etwas eigenwillige Deklaration (1==1), die eben immer TRUE ist!)

```
/* bitfields.c */
#include <stdio.h>
#define TRUE    (1==1)
#define FALSE   (0==1)

void main(void)
{
    struct Bitfields
    {
        unsigned int optionA : 1;
        unsigned int optionB : 1;
        unsigned int optionC : 1;
    } bf;

    printf("\nstruct Bitfields benötigt %d Bytes.\n",
        sizeof(struct Bitfields));
    bf.optionA=TRUE;
    bf.optionB=TRUE;
    bf.optionC=FALSE;
    printf("Option A ist%s aktiv.\n", (bf.optionA ? " " : " nicht"));
    printf("Option B ist%s aktiv.\n", (bf.optionB ? " " : " nicht"));
    printf("Option C ist%s aktiv.\n", (bf.optionC ? " " : " nicht"));

    bf.optionA = ~bf.optionA;    /* Erinnern Sie sich noch an ~? */

    printf("\nNach \"bf.optionA = ~bf.optionA;:\n");
    printf("Option A ist%s aktiv.\n", (bf.optionA ? " " : " nicht"));
    printf("Option B ist%s aktiv.\n", (bf.optionB ? " " : " nicht"));
    printf("Option C ist%s aktiv.\n", (bf.optionC ? " " : " nicht"));

} /* end main */
```

²⁴Die Bit-Felder haben nichts mit einer bekannten Biersorte zu tun.

Ablauflisting:

```
struct Bitfields benötigt 4 Bytes.  
Option A ist aktiv.  
Option B ist aktiv.  
Option C ist nicht aktiv.
```

```
Nach "bf.optionA = ~bf.optionA;":  
Option A ist nicht aktiv.  
Option B ist aktiv.  
Option C ist nicht aktiv.
```

Es wird lokal in main() ein Bitfeld bzw. eine Struktur Bitfields deklariert, die über drei 1-Bit-Komponenten (optionA, optionB, optionC) verfügt. Wie das Programm zeigt, wird rechnerabhängig für die Bitfeld-Variable bf die nächsthöhere ganzzahlige Grundeinheit genommen, hier 4 Bytes auf einer UNIX-Anlage. Auf einem PC werden in dieser Situation üblicherweise zwei Bytes verwendet.

8.6. Pointer

Ein Pointer (Zeiger) ist ein Datentyp, bei dem Speicheradressen verwaltet werden. Eine Variable von einem Pointertyp kann also jeweils eine konkrete Speicheradresse (z.B. von einer anderen statischen Variablen) beinhalten. Pointer sind insoweit dynamisch, als sie mit ihrer Deklaration zunächst keinen weiteren Speicherplatz zugewiesen bekommen als den für die eigentliche Adresse.

Die allgemeine Syntax der Pointerdeklaration hat die Grundform:

```
<datentyp> * <bezeichner>;
```

Und wieder ist es wie bei Pascal: die Pointervariable selbst gehorcht den üblichen Gültigkeitsregeln wie alle anderen Variablen auch; der Speicherplatz, auf den sie aktuell zeigt, kann sich jedoch *irgendwo* im Arbeitsspeicher befinden!

Beispiele:

```
int *pi;          /* pi ist ein Zeiger auf einen int-Speicherplatz */  
char *s;         /* s ist ein Zeiger auf einen char-Speicherplatz, */  
                /* damit jedoch bereits als Zeichenkette nutzbar! */  
float *pf;       /* pf ist entsprechend ein Pointer auf float */
```

Sehen wir uns ein erstes kleines Beispielprogramm an, in dem auch von der sogenannten Pointerarithmetik Gebrauch gemacht wird: zu einem Pointer dürfen wir int-Werte addieren (und von ihm in den definierten Grenzen auch abziehen), auf eigenes Risiko, denn wir müssen dabei selbst aufpassen, daß wir in erlaubtem Speicherbereich bleiben.²⁵

²⁵Daß die Speicheradressen, die hier auftauchen, so viel kleiner sind als die z.B. in arrays1.c aufgetretenen Werte, liegt daran, daß dieses Beispiel hier auf einem PC protokolliert wurde: die Speicheradressen sind natürlich systemabhängig!

```

/* pointer1.c */
#include <stdio.h>

void main(void)
{
    int dummy=99, i;
    char *s, c;
    int *pi, a[3]={ 100,110,120 } ; /*Initialisierung eines Arrays*/

    printf("1.Teil:\n");
    c='A';
    s=&c;                                     /* & ist der Adreßoperator */
    printf("s=%s  s=%d  &c=%d\n",s,s,&c);

    printf("2.Teil:\n");
    pi=a;
    printf("pi=%d  *pi=%d  *pi+1=%d  *(pi+1)=%d\n",
        pi,*pi,*pi+1,*pi+1);
    pi++;
    printf("pi=%d  *pi=%d  *pi+1=%d  *(pi+1)=%d\n",
        pi,*pi,*pi+1,*pi+1);

    printf("3.Teil:\n");
    i=100;
    pi=&i;
    printf("pi=%d  *pi=%d  *pi+1=%d  *(pi+1)=%d\n",
        pi,*pi,*pi+1,*pi+1);
} /* end main */

```

Ablauflisting:

```

1.Teil:
s=A$Æf  s=3678  &c=3678
2.Teil:
pi=3670  *pi=100  *pi+1=101  *(pi+1)=110
pi=3672  *pi=110  *pi+1=111  *(pi+1)=120
3.Teil:
pi=3682  *pi=100  *pi+1=101  *(pi+1)=99

```

Zu dem Programm im einzelnen: Zunächst werden zwei int-Variablen deklariert (dummy und i), wobei dummy bereits auf 99 initialisiert wird. Dann werden ein Zeiger auf char (s) und ein char (c), ein Zeiger auf int (pi) und ein int-Array mit drei Komponenten (a) vereinbart. Das Array erhält bereits bei der Deklaration seine Startwerte: a[0]=100, a[1]=110 und a[2]=120.

Im 1.Teil wird dann c auf 'A' und der Zeiger s auf die Adresse von c (&c) gesetzt. Im Ablauflisting sehen wir denn auch, daß s[0] nun den Wert 'A' besitzt, die Adressen s und &c sind auch tatsächlich gleich (hier: 3678). Allerdings sehen wir bei der Textausgabe von s, daß nach dem 'A' noch aller möglicher Schrott (hier beispielhaft: \$Æf) folgt: das liegt daran, daß Zeichenketten soweit reichen, bis '\0' gefunden wird!

Im 2.Teil wird der Zeiger pi auf a, d.h. die Adresse des Arrays gesetzt. (Wir erinnern uns: Arrays sind bereits die Adressen!) Im darauffolgenden printf werden der Reihe nach ausgegeben: der Inhalt des Pointers pi (3670), der Inhalt des Speicherplatzes (3670), auf den

pi zeigt (*pi, hier: 100), dann dieser Wert plus 1 (101) und zuletzt der Inhalt des nächsten Speicherplatzes, hier 110, denn pi zeigte ja auf das erste Element des Arrays a, und 110 ist der zweite Array-Eintrag; pi+1 ist die oben erwähnte Pointerarithmetik. Anschließend wird mit pi++; pi einen Speicherplatz weitergesetzt, das ist nicht 1 Byte, sondern 1*sizeof(int)! Die printf()-Ausgabe der entsprechenden Werte erklärt sich fast selbst.

Im 3. Teil schließlich erhält i den Wert 100, pi wird auf die Adresse von i (hier ist das 3682) gesetzt. Und als Überraschung und Warnung gleichzeitig: mit (pi+1) stoßen wir diesmal auf einen Speicherplatz, der mit i natürlich nichts mehr zu tun hat: dort steht der Wert *(pi+1)=99, der zuvor auf die dummy-Variable zu Kontrollzwecken zugewiesen wurde. Sie sehen: C läßt uns eine ganze Menge Freiheit!

8.7. Pointer auf Pointer und Arrays von Pointern

Mit Pointern kann man viel machen. Man kann sie zum Beispiel auch iterieren (Zeiger auf Zeiger). Ebenso können Arrays von Pointern und Pointer auf Arrays deklariert werden und vieles mehr. Im folgenden sollen beispielhaft einige Aspekte diskutiert werden, die für die weitere Praxis der C-Programmierung relevant sein können.

Beispiel: Betrachten wir die folgenden Deklarationen.

```
char **bildschirm;  
char (*bs2)[80];  
char *bs3[80];  
char *(bs4[80]);  
char bs5[25][80];
```

Hier ist bildschirm ein Zeiger auf Zeiger von char; solch eine Variable könnte z.B. dazu benötigt werden, einen Bildschirminhalt zu verwalten, wobei die Bildschirmgröße zur Compilezeit nicht zwingend feststehen muß²⁶.

bs2 ist ein Pointer auf ein Array von 80 Zeichen; bs3 und bs4 sind identische Arrays von jeweils 80 Zeigern auf char; bs5 schließlich ist ein harmloses zweidimensionales Array von Zeichen.

Das nachfolgende Beispiel illustriert den nicht ganz trivialen Umgang mit Pointern auf Pointer am Beispiel einer Variablen bildschirm, die in der Praxis dazu dienen kann, den jeweiligen Bildschirminhalt zu verwalten.

Mit der Deklaration char** bildschirm; gibt es indes erst einen einzigen Speicherplatz! Dieser ist so groß, wie auf dem jeweiligen System Pointer(variablen), Adressen eben, sind. Um wie hier fünfundzwanzig Zeilen verwalten zu können, muß zunächst für diese fünfundzwanzig (Zeilen-)Pointer Speicherplatz allokiert werden, was mit der Bibliotheksroutine malloc() (memory allocate, Prototyp in stdlib.h) geschehen kann. Bei malloc() ist zum einen anzugeben, wieviel Speicherplatz benötigt wird, in der Regel geschieht dies mit der Verwendung von sizeof(), und zum anderen, für was für einen Typ der Pointer dienen soll. In unserem Fall benötigt bildschirm ZEILEN-mal die Größe eines Pointers auf char (das ist sizeof(char*)), und bildschirm selbst ist ein Zeiger auf Zeiger auf char. (Das steht ja auch in der Deklaration von bildschirm.) Dies sieht dann konkret so aus:

²⁶Vgl. hierzu das nachfolgende Programmbeispiel pointer2.c.

```
bildschirm=(char **)malloc(sizeof(char*)*ZEILEN);
```

Jetzt stehen ZEILEN-viele Pointer auf char zur Verfügung. Nun kann für jede Zeile Speicherplatz für einen solchen Pointer auf char allokiert werden, der SPALTEN-viele Zeichen aufnehmen muß; für i von 0 bis ZEILEN-1 ist also folgendes notwendig:

```
bildschirm[i]=(char *)malloc(sizeof(char)*SPALTEN);
```

Die so bereitgestellten Speicherplätze stehen solange im Programmablauf zur Verfügung, bis sie (oder Teile davon) mit free() wieder freigegeben werden. free() ist ebenfalls eine in stdlib.h deklarierte Bibliotheksfunktion.

```
/* pointer2.c */

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

#define SPALTEN 81    /* Diese Werte könnten auch dynamisch erst */
#define ZEILEN 25    /* festgelegt werden als Variablen!          */

void main(void)
{
    char **bildschirm;
    int i,j;

    /* Speicherplatz für die ZEILEN viele Zeilenpointer wird geholt*/
    bildschirm=(char **)malloc(sizeof(char*)*ZEILEN);
    if (bildschirm==NULL) /* malloc() scheiterte! */
    {
        fprintf(stderr, "\nFehler bei malloc()!\n");
        exit(EXIT_FAILURE); /* einfachste Fehlerbehandlung! */
    }

    for (i=0; i<ZEILEN; i++)
    {
        /* Speicherplatz für jede Zeile wird geholt */
        bildschirm[i]=(char *)malloc(sizeof(char)*SPALTEN);
        if (bildschirm[i]==NULL) /* malloc() scheiterte! */
        {
            fprintf(stderr, "\nFehler bei malloc()!\n");
            exit(EXIT_FAILURE);
        }
        /* Sicherheitshalber eine sehr umsichtige Initialisierung */
        for (j=0; j<SPALTEN; j++)
            bildschirm[i][j]='?';
        /* Kopieren eines Mustertextes in die i-te Zeile */
        strcpy(bildschirm[i], "Mustertext");
    }

    /* Kontrollausgabe der Zeilen */
```

```

for (i=0; i<ZEILEN; i++)
{
    printf("\n%d->:%s:",i,bildschirm[i]);
    /*Alternative dazu: printf("\n%d->:%s:",i,*(bildschirm+i));*/
}

/* Und hier wird der gesamte Bildschirm zeichenweise ausgegeben*/
printf("\n");
for (i=0; i<ZEILEN; i++)
{
    for (j=0; j<SPALTEN-1; j++)
    {
        putchar(bildschirm[i][j]);
    }
    putchar('\n');
}
/* Der Speicherplatz wird wieder freigegeben: zuerst der
für alle Zeilen und danach der für die Zeilenpointer */
for (i=0; i<ZEILEN; i++)
    free(bildschirm[i]);
free(bildschirm);
} /* end main */

/* end of file pointer2.c */

```

8.8. Kommandozeilenargumente und Umgebungsinformation

ANSI C berücksichtigt die Tatsache, daß in der Praxis zum einen sehr häufig beim Aufruf Argumente direkt an das auszuführende Programm mitgegeben werden müssen, zum zweiten, daß Informationen aus dem Umgebungsbereich (*environment*) verarbeitet werden müssen. Auf die Möglichkeiten, die C hier bietet, soll in diesem Abschnitt eingegangen werden.

8.8.1 Kommandozeilenargumente: argc und argv

Bisher war die Deklaration der (Hauptprogramm-)Funktion main() stets

```
void main(void
```

- das heißt: an main() wurde bisher nichts übergeben. Hier sieht C über die Deklaration

```
void main(int argc, char **argv)
```

oder, synonym dazu,

```
void main(int argc, char *argv[])
```

die Möglichkeit vor, eine Anzahl von Parametern (als Anzahl von Zeichenketten) beim Programmaufruf mitzugeben. Der Name argc steht dabei für den *argument counter*, den Zähler, argv (*argument vector*) ist das Array (oder der Zeiger) auf die argc vielen Zeichenketten, der sogenannte argument vector.

Wird das Programm argument beispielsweise aufgerufen in der Form

```
argument 1 2 3
```

so sind `argc==4 (!)`, `argv[0]=="argument\0"`, `argv[1]=="1\0"`, `argv[2]=="2\0"` und – nun erraten Sie es bereits – `argv[3]=="3\0"`.

Ein minimales Beispiel hierzu finden Sie im nächsten Unterabschnitt.

8.8.2 Umgebungsinformation: envp

Die Informationen über den Umgebungsbereich können gegebenenfalls über einen dritten Parameter bei `main()` abgerufen werden. Mit der Deklaration

```
void main(int argc, char **argv, char **envp)
```

oder, synonym dazu,

```
void main(int argc, char *argv[], char *envp[])
```

kann mit der Variablen `envp` (*environment pointer*) ein Zeiger auf den Umgebungsbereich angelegt werden. Dies funktioniert in entsprechender Weise bei den beiden Betriebssystemen UNIX und DOS.

Die Verwendung von `envp` (und der Parameter `argc` und `argv`) soll das nachstehende kleine Beispiel illustrieren.

```
/*
   envp.c
   Kurzdemonstration: Kommandozeilenargumente inkl. Umgebungsbereich
   */

#include <stdio.h>

void main(int argc, char **argv, char **envp)
{
    int i, nr=1;
    for (i=0; i<argc; i++)
        printf("Argument[%d]=[s]\n",i,argv[i]);

    printf("\nUmgebungsbereich:\n");
    while (*envp)
        printf("%2d: \"%s\"\n",nr++,*(envp++));
} /* end main */
```

Nachstehend noch die Ablauflistings von `envp` bzw. `ENVP.EXE` – einmal unter UNIX, einmal unter DOS.

Bildschirmprotokoll von `envp.c` unter UNIX:

Aufruf des Programms:

```
envp argument1 argument2
```

Ausgabe des Programms:

```
Argument[0]=[envp]
```

```
Argument[1]=[argument1]
```

```
Argument[2]=[argument2]
```

Umgebungsbereich:

```
1: "_=./envp"
2: "LANG=n-computer"
3: "NLSPATH=/usr/lib/nls/n-computer/%N.cat:/usr/lib/nls/C/%N.cat"
4: "PATH=/bin:/usr/bin:/usr/contrib/bin:/usr/local/bin:."
5: "COLUMNS=80"
6: "SHELL_LEVEL=1"
7: "EDITOR=/usr/contrib/bin/me"
8: "CCOPTS=-Aa"
9: "HISTFILE=/users/guest/.sh_history"
10: "LOGNAME=guest"
11: "MAIL=/usr/mail/guest"
12: "NETID=pcr103.bg.bib.de"
13: "SHELL=/bin/ksh"
14: "HISTORY=100"
15: "TMOUT=1200"
16: "HOME=/users/guest"
17: "FCEDIT=vi"
18: "TERM=vt220-ncsa"
19: "PWD=/tmp"
20: "TZ=MEZ-1MESZ"
```

Bildschirmprotokoll von envp.c unter DOS:

Aufruf des Programms:

```
envp argument1 argument2
```

Ausgabe des Programms:

```
Argument[0]=[D:\C\STUFF\ENVP.EXE]
```

```
Argument[1]=[argument1]
```

```
Argument[2]=[argument2]
```

Umgebungsbereich:

```
1: "COMSPEC=c:\4dos\4dos.com"
2: "TEMP=c:\tmp"
3: "OLDPATH=d:\bat;c:\bat;c:\dos;c:\sw\bib;c:\util;c:\novell\netware4;x:\util"
4: "NETNAME=PCR103"
5: "TMP=c:\tmp"
6: "LOGNAME=BAEUMLE"
7: "PLANPATH=H:\VERW\INFO\PLAN"
8: "WINPMT=[Windows aktiv $t$h$h$h] $p$g$s"
9: "PATH=c:\4dos;c:\dos;c:\sw\bib;c:\sw\util;c:\novell;c:\bat"
```

8.9. Rekursion und Rekursive Strukturen: Listen und Bäume

Wie in jeder anständigen Programmiersprache²⁷ ist Rekursion auch in C möglich, gleichermaßen direkte Rekursion wie indirekte. In den folgenden Abschnitten sollen mit zunehmendem Schwierigkeitsgrad drei Beispiele für Rekursion in C vorgestellt werden.

8.9.1 Rekursion

Das Grundprinzip der direkten Rekursion in C: eine Funktion $f()$ ruft sich selbst auf, sofern nicht eine Abbruchbedingung (Rekursionsboden) erfüllt ist. Die indirekte Rekursion unterscheidet sich davon dadurch, daß zwei oder mehr Funktionen beteiligt sind, die sich wechselseitig aufrufen.

Zur einfachen, direkten Rekursion sei das beliebte Beispiel der Berechnung der Fakultät angeführt: zu einer Zahl n soll $n!$, das Produkt aller natürlichen Zahlen von 1 bis n , berechnet werden. – Nachfolgend ein Programmlisting hierzu.

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

long fakultaet(long);

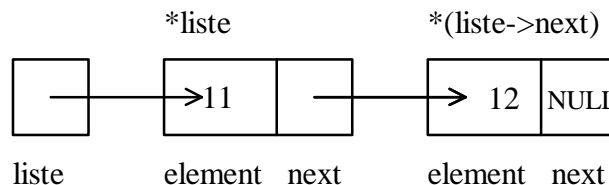
void main(void)
{
    long n=5;
    printf("\nDie Fakultät von %ld ist %ld.\n",n,fakultaet(n));
} /* end main */

long fakultaet(long n)
{
    if (n>1)
        return n*fakultaet(n-1);
    return 1;
} /* end fakultaet */
```

²⁷So erlauben etwa auch Pascal oder Modula-2, nicht jedoch das klassische COBOL, rekursive Programmstrukturen.

8.9.2 Lineare Listen

Eine lineare Liste ist eine (dynamische) Datenstruktur, die, über Zeiger verknüpft, beliebig (endlich) viele Werte (eines gewissen Typs) z.B. geordnet verwalten kann.



Wird in eine solche, aufsteigend geordnete, lineare Liste (von int-Werten beispielsweise) ein neuer Wert aufgenommen, so muß dafür ein neuer Speicherplatz mit `malloc()` allokiert und der entsprechende Eintrag an die passende Stelle der bisherigen Liste eingefügt werden; ist der neue Wert der größte, so muß der neue Speicherplatz einfach an das Ende der Liste angehängt werden. Ist die Liste (noch) leer, so bildet dieser Wert zusammen mit dem entsprechenden Zeiger die komplette lineare Liste.

Das nachstehende Programm `linliste.c` zeigt beispielhaft, wie auf der Kommandozeile eine Reihe von int-Werten mitgegeben werden, die zugehörige sortierte lineare Liste aufgebaut und dann wieder ausgegeben wird.

Anmerkungen zu dem Programm:

- Die Datenstruktur `struct LineareListe` bzw. `LISTE` beinhaltet der Einfachheit halber nur die zwei Komponenten `element` (`int`) und `next` (`LISTE*`), einen Zeiger auf das nächste Listenelement. Dieser hat den Wert `NULL`²⁸, wenn kein Nachfolger in der Liste existiert, dies also der letzte Eintrag in der Liste ist.
- Wie für gute Programme üblich: wird `linliste` ohne die korrekte Anzahl Parameter, hier also ohne Parameter, aufgerufen, so wird eine kurze Erläuterung ausgegeben, wie das Programm korrekt aufgerufen werden muß.
- Das schöne Konstrukt `while(*(++argv))` testet, ob `*argv!=NULL` ist, d.h. ob noch weitere Parameter im Argumentenvektor `argv` vorhanden sind; `++argv` bewirkt, daß dieser Zeiger bei jedem Schleifendurchlauf weitergesetzt wird. Hier muß der Präfixoperator `++` genommen werden, damit nicht `argv[0]` (der Programmname selbst) mit verarbeitet wird.
- `atoi()` ist eine Konvertierungsfunktion der Standardbibliothek: `atoi` steht für `ascii-to-int` und wandelt eine Zeichenkette um in einen `int`-Wert. (Dazu verwandte Funktionen sind `atof()`, `atod()` usw.)

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

²⁸Dem `NULL` entspricht bei Pascal `NIL`.

```

#include <stdlib.h>

/* Struktur und deren Typ deklarieren */
typedef struct LineareListe
{
    int          element;
    struct LineareListe *next;
} LISTE;

/* Prototypen */
void Einfuegen(LISTE **,int);
void Ausgeben(LISTE *);

/* Hauptprogramm */
int main(int argc, char **argv)
{
    LISTE *liste;
    liste=NULL;

    /* Falls keine Parameter angegeben: kurze Erläuterung geben */
    if (argc==1)
    {
        fprintf(stderr,"\nAufruf: linliste intwert "
            "{ intwert ... }\n");
        return(EXIT_FAILURE);
    } /* end argc==1, d.h. keine Parameter */

    /* Die Werte werden in die lineare Liste eingefügt */
    while (*(++argv))
        Einfuegen(&liste,atoi(*argv));

    /* Die lineare Liste wird zur Kontrolle ausgegeben */
    Ausgeben(liste);

    /* An das Betriebssystem zurückgeben: alles ok! */
    return(EXIT_SUCCESS);
} /* end main */

/* Einfuegen von wert an passender Stelle (aufsteigend sortiert) */
void Einfuegen(LISTE **pliste,int wert)
{
    if (*pliste==NULL)
    {
        *pliste = (LISTE*)malloc(sizeof(LISTE));
        if (*pliste == NULL)
        {
            fprintf(stderr,"\nmalloc()-Aufruf schlug fehl!\n");
            exit(EXIT_FAILURE);
        } /* end if malloc() schlug fehl */
        (*pliste)->element=wert;
    }
}

```

```

    (*pliste)->next=NULL;
} /* end if *pliste==NULL */
else if (wert < (*pliste)->element) /* hier einfügen */
{
    LISTE *ptr2;
    int tmp;
    ptr2=(LISTE*)malloc(sizeof(LISTE));
    if (ptr2 == NULL)
    {
        fprintf(stderr, "\nmalloc()-Aufruf schlug fehl!\n");
        exit(EXIT_FAILURE);
    } /* end if malloc() schlug fehl */
    /* Auf dem neuen Platz wird der alte Listeneintrag */
    /* gespeichert - und dort wird der neue Wert eingetragen */
    tmp=(*pliste)->element;
    (*pliste)->element=wert;
    ptr2->element=tmp;
    /* Nun wird der neue Wert an der aktuellen Position einge-*/
    /* schoben, der Rest der Liste nach hinten gehängt. */
    ptr2->next=(*pliste)->next;
    (*pliste)->next=ptr2;
} /* Position zum Einfügen gefunden */
else
{
    Einfuegen(&((*pliste)->next),wert);
} /* rekursiver Zweig - weiter hinten anhängen oder einfügen*/
} /* end Einfuegen */

/* Einfache Ausgabe der linearen Liste */
void Ausgeben(LISTE *liste)
{
    if (liste==NULL)
        printf(" (Ende der Liste)\n");
    else
    {
        printf("%d ",liste->element);
        Ausgeben(liste->next);
    }
} /* end Ausgeben */
/* Ende der Datei linliste.c */

```

Aufruf des Programms:

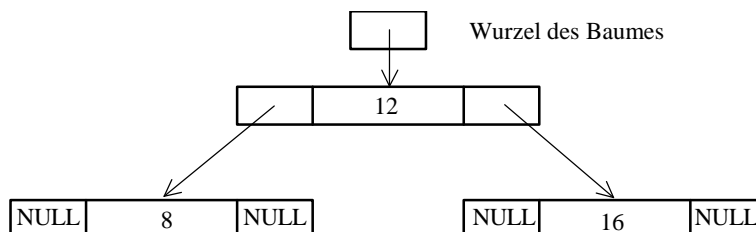
```
linliste 3 7 1 6 4 2 5 8 9 10
```

Ausgabe des Programms:

```
1 2 3 4 5 6 7 8 9 10 (Ende der Liste)
```

8.9.3 Bäume

Lineare Listen sind ein Spezialfall von Bäumen. Ein *Baum* ist eine dynamische Datenstruktur mit einzelnen Elementen (Knoten) und einer Ordnung mit Vorgänger und Nachfolger, so daß jeder Knoten (außer dem ersten, der sogenannten *Wurzel*) genau einen (direkten) Vorgänger und endliche viele Nachfolger besitzt. Ist die Anzahl der Nachfolger auf 2 beschränkt, so spricht man von einem binären Baum.



Das nachstehende – nicht mehr ganz so – kleine Programm `baum1.c` illustriert exemplarisch die Deklaration von und den Umgang mit binären Bäumen:

Die Funktion `InsertTree()` trägt einen neuen Wert (zwischen 1 und 99) so in den Baum ein, daß dieser gemäß der Inorder-Sortierung aufgebaut wird, das bedeutet, daß links unterhalb eines Knotens nur kleinere, rechts unterhalb eines jeden Knotens nur größere Werte abgespeichert werden. Vgl. hierzu das Ablauflisting zum folgenden Beispielprogramm.

Die Funktion `DisplayTree()` gibt den jeweiligen Baum mit einfacher Liniengraphik dargestellt auf den Bildschirm aus.

Die Funktion `Deallocate()` schließlich gibt den für den Baum per `malloc()` reservierten Speicherplatz wieder frei.

Sämtliche Funktionen sind der Natur der Sache angemessen rekursiv formuliert, denn der Datentyp `TREE` ist ebenfalls eine rekursive Struktur!

```
/*
  baum1.c
  Demonstration: Rekursion innerhalb eines binären Baumes.
*/

#include <stdio.h>

/** Symbolische Konstanten **/
#define ZEILEN 6
#define SPALTEN 80

/** Typendeklaration für TREE und AUSGABE **/
typedef struct BinaryTree
{
    int root;
    struct BinaryTree *left, *right;
} TREE;

typedef char AUSGABE[ZEILEN][SPALTEN];
```

```

/** Prototypen */
void Deallocate(TREE*);
void InsertTree(TREE**,int);
void DisplayTree(TREE *,AUSGABE,int,int,int);

void main(void)
{
    TREE    *t;
    int     wert, i, j;
    AUSGABE buffer;

    /* Initialisierungen: Bildschirm-Buffer und TREE-Pointer t */
    t=NULL;
    for (i=0; i<ZEILEN; i++)
    {
        for (j=0; j<SPALTEN-1; j++)
            buffer[i][j]=' ';
        buffer[i][SPALTEN-1]='\0';
    } /* end for i */

    /* Ein-/Ausgabeschleife */
    do {
        printf("\nAktueller Inhalt des Baumes: ");
        if (t) {
            DisplayTree(t,buffer,SPALTEN,0,SPALTEN/2);
            for (i=0; i<ZEILEN; i++)
                printf("\n%s",buffer[i]);
        } else {
            printf(" (leer) \n\n");
        } /* end if (t) */
        printf("\nWelche ganze Zahl im Bereich 1..99 soll "
            "aufgenommen werden?\n[0=Ende] >");
        scanf("%d",&wert);
        if (wert<=0 || wert>=100)
            break;
        InsertTree(&t,wert);
    } while (wert);
    Deallocate(t);
} /* end main */

void Deallocate(TREE * tree)
{
    if (tree==NULL) /* Ist der Baum leer? Dann Abbruch... */
        return;
    if (tree->left!=NULL) /* Ansonsten erst den linken, dann */
        Deallocate(tree->left);
    if (tree->right!=NULL) /* den rechten Teilbaum löschen, */
        Deallocate(tree->right);
    free(tree); /* dann den aktuellen Knoten. */
} /* end Deallocate */

```

```

void InsertTree(TREE **pt, int wert)
{
    if (*pt==NULL)
    {
        *pt=(TREE*)malloc(sizeof(TREE));
        (*pt)->left=(*pt)->right=NULL;
        (*pt)->root=wert;
    }
    else
    {
        /* es werden nur neue Werte eingetragen! */
        if (wert > (*pt)->root) /* rechts anhängen */
            InsertTree(&((*pt)->right),wert);
        else if (wert < (*pt)->root) /* links anhängen */
            InsertTree(&((*pt)->left),wert);
    } /* end if *pt */
} /* end InsertTree */

void DisplayTree(TREE * t,AUSGABE buffer,
                int orient, int zeile, int spalte)
/* Angedeutete graphische Ausgabe des Baumes; die vorliegende
 * Variante verarbeitet nur maximal zweistellige positive
 * int-Einträge und geht bei einem 80-Spalten-Bildschirm
 * höchstens bis zu einer Tiefe von 6 mit einer akzeptablen
 * Darstellung.
 */
{
    /* linken Teilbaum ausgeben */
    if (t->left!=NULL && zeile<ZEILEN-1)
    {
        int i;
        DisplayTree(t->left,buffer,spalte, zeile+1,
                    spalte - abs(orient-spalte)/2 );
        /* Eine Prise Liniengraphik ... */
        for (i=spalte-abs(orient-spalte)/2; i<spalte; i++)
            buffer[zeile][i]='-';
        buffer[zeile][spalte-abs(orient-spalte)/2]='+';
    }
    /* rechten Teilbaum ausgeben */
    if (t->right!=NULL && zeile<ZEILEN-1)
    {
        int i;
        DisplayTree(t->right,buffer,spalte, zeile+1,
                    spalte + abs(orient-spalte)/2 );
        /* Eine Prise Liniengraphik ... */
        for (i=spalte+abs(orient-spalte)/2; i>spalte; i--)
            buffer[zeile][i]='-';
        buffer[zeile][spalte+abs(orient-spalte)/2]='+';
    }
    /* diesen Knoten ausgeben: Ausgabe zuletzt, damit
     * die Liniengraphik ggf. überschrieben wird!
     */
}

```

```

if (t->root > 9)
    buffer[zeile][spalte-1] = (t->root)/10 + '0';
    buffer[zeile][spalte]   = (t->root)%10 + '0';
} /* end DisplayTree */

```

Ablauflisting:

Aktueller Inhalt des Baumes: (leer)

/** Listing hier gekürzt **/

Welche ganze Zahl im Bereich 1..99 soll aufgenommen werden?

[0=Ende] >66

Aktueller Inhalt des Baumes:

```

          +-----45-----+
+-----23-----+          +-----75-----+
12                25          66                78----+
                                                99

```

Welche ganze Zahl im Bereich 1..99 soll aufgenommen werden?

[0=Ende] >77

Aktueller Inhalt des Baumes:

```

          +-----45-----+
+-----23-----+          +-----75-----+
12                25          66                +---78----+
                                                77          99

```

Welche ganze Zahl im Bereich 1..99 soll aufgenommen werden?

[0=Ende] >0

9 Dateiverarbeitung

In ANSI C werden Dateien (*files*) über einen Dateizeiger (*File-Pointer*) vom Typ FILE verwaltet, der in der Headerdatei `stdio.h` deklariert ist²⁹.

9.1. Vordefinierte Dateien `stdin`, `stdout` und `stderr`

Auch ohne in C explizit auch nur eine Datei deklariert zu haben, sind drei Dateien (bzw. File-Pointer darauf) bereits geöffnet: der Standardeingabekanal `stdin` (Default: Tastatur), die Standardausgabe `stdout` (Default: Bildschirm) und der Fehlerkanal `stderr` (Default: Bildschirm). Selbstverständlich können beim Aufruf eines Programms diese Standardkanäle umgelenkt werden. Unter dem Betriebssystem UNIX³⁰ kann das Programm `a.out` aufgerufen werden z.B. in der Form

```
a.out > outfile < infile 2> errorfile
```

Damit ist dann `stdin` ein Pointer auf die Datei `infile`, `stdout` einer auf `outfile` und `stderr` ein Pointer auf die Datei `errorfile`.

²⁹Die konkrete Gestalt des Datentyps FILE sieht bei jedem Compiler(hersteller) verschieden aus. Um sich ein Bild davon machen zu können wird nachstehend die FILE-Deklaration des Turbo C++ 3.0 Compilers von Borland gezeigt.

```
typedef struct
{
    int          level;          /* fill/empty level of buffer */
    unsigned     flags;         /* File status flags          */
    char         fd;           /* File descriptor            */
    unsigned char hold;        /* Ungetc char if no buffer  */
    int          bsize;        /* Buffer size                 */
    unsigned char *buffer;     /* Data transfer buffer       */
    unsigned char *curp;       /* Current active pointer     */
    unsigned     istemp;       /* Temporary file indicator   */
    short        token;        /* Used for validity checking */
} FILE;                       /* This is the FILE object   */
```

Bei dem Hewlett-Packard HP-UX C++ Compiler zur UNIX-Betriebssystemversion 9.04 sieht die Deklaration von FILE so aus:

```
typedef struct
{
    int          __cnt;
    unsigned char* __ptr;
    unsigned char* __base;
    unsigned short __flag;
    unsigned char __fileL;     /* low byte of file desc */
    unsigned char __fileH;     /* high byte of file desc */
} FILE;
```

³⁰Dies funktioniert auch beim PC-Betriebssystem MS-DOS in analoger Weise.

9.2. Sequentieller Dateizugriff und I/O-Funktionen

Die sequentielle Dateiverarbeitung in C läuft nach dem Grobmuster ab: Öffnen der Datei mit `fopen()`, Verarbeiten der Daten(sätze) mit verschiedenen Bibliotheksroutinen, Schließen der Datei mit `fclose()`. Dabei kann in zwei Modi gearbeitet werden: als Textdatei (bei den meisten Compilern der Default) oder als Binärdatei. Der Unterschied ist der, daß bei Binärdateien keinerlei Konvertierungen beim Lesen aus oder Schreiben in Dateien stattfinden, während bei Textdateien betriebssystemspezifische Umwandlungen erfolgen können. So ist unter MS-DOS das Zeilenende als `"\r\n"` (Carriage Return + Line Feed) definiert, während es in C (und unter UNIX) nur `"\n"` (Line Feed) ist. Hier wird beim Arbeiten im Textmodus automatisch eine Umwandlung vorgenommen.

Die Prototypen der hier vorgestellten Funktion befinden sich im `stdio.h`-Headerfile.

Ablaufschema:

- 1.Schritt: Öffnen der Datei mit `fopen()`
- 2.Schritt: Verarbeiten der Datensätze, das heißt:
 - a) zeichenweise Ein-/Ausgabe (`fgetc()`, `fputc()`)
 - b) zeilenweise Ein-/Ausgabe (`fgets()`, `fputs()`) (Textfiles)
 - c) formatierte Ein-/Ausgabe (`fscanf()`, `fprintf()`)
 - d) blockweise Ein-/Ausgabe (`fread()`, `fwrite()`)
- 3.Schritt: Schließen der Datei mit `fclose()` [mittelbar bei `exit()`]

Wir wollen dies im folgenden schrittweise ausformulieren.

9.2.1 Öffnen der Datei

Prototyp:

```
FILE * fopen(const char * filename, const char * mode);
```

Aktion: `fopen()` versucht die Datei `filename` zu öffnen in dem unter `mode` beschriebenen Modus.

Dabei kann `mode` folgendes sein:

- "r" für lesenden Zugriff auf eine existente Datei,
- "w" für schreibenden Zugriff (destruktives Schreiben),
- "a" für anhängenden Schreibzugriff auf eine bel. Datei;

Dahinter kann (optional) mit einem "t" oder "b" explizit gesagt werden, ob im Text- oder Binärmodus gearbeitet werden soll. Dahinter wiederum kann mit einem "+" angegeben werden, daß lesend und schreibend zugegriffen werden soll!

"w+" bedeutet, daß lesend und schreibend auf eine eventuell schon existente Datei zugegriffen werden soll, wobei allerdings die Dateilänge zuerst auf 0 gesetzt wird, d.h. es handelt sich wiederum um ein destruktives Schreiben.

"a+" schließlich öffnet die Datei so, daß an das Ende der evtl. bereits existenten Datei geschrieben wird.

Rückgabe: `fopen()` liefert einen FILE-Pointer zurück oder NULL, falls der Zugriff scheitert.

Beispiel:

```
if ((fp=fopen("beispiel","wb"))==NULL)
{
    fprintf(stderr,"Fehler beim Öffnen der Datei\n");
    exit(EXIT_FAILURE);
} /* Öffnen der Datei beispiel zum binären Schreiben */
```

9.2.2 Verarbeiten der Datensätze

9.2.2.1. Zeichenweise Ein-/Ausgabe

Prototyp:

```
int fgetc(FILE *fp);
```

Aktion: `fgetc()` holt das nächste Zeichen aus der Datei `fp`.

Rückgabe: Die Funktion liefert den ASCII-Wert des aktuellen Zeichens aus der korrekt geöffneten Datei mit dem FILE-Pointer `fp` zurück oder EOF im Fehlerfalle oder am Ende der Datei.

Prototyp:

```
int fputc(int c, FILE *fp);
```

Aktion: Das Zeichen `c` wird von `fputc()` in die Datei mit dem FILE-Pointer `fp` geschrieben.

Rückgabe: Das Zeichen `c` wird (im Sinne des numerischen Wertes) zurückgeliefert, EOF im Fehlerfalle.

9.2.2.2. Zeilenweise Ein-/Ausgabe bei Textdateien

Prototyp:

```
char *fgets(char *s, int length, FILE *fp);
```

Aktion: Es wird bis EOF, bis `\n` oder zum Erreichen der angegebenen Länge `length` (bzw. `length-1`) aus der Datei `fp` gelesen und in den Buffer `s` geschrieben, der genügend Platz bereitgestellt haben muß.

Rückgabe: Im Erfolgsfall wird ein Pointer auf `s` zurückgeliefert, im Fehlerfalle ein NULL-Pointer.

Beispiel:

```
fgets(buffer,128,fp);
```

Prototyp:

```
int fputs(char *s, FILE *fp);
```

Aktion: Die Zeichenkette `s` wird in die Datei mit dem FILE-Pointer `fp` geschrieben. Das Stringendezeichen `\0` wird dabei nicht in die Datei übernommen.

Rückgabe: `fputs()` liefert die Anzahl der übertragenen Zeichen zurück, im Fehlerfalle EOF.

Beispiel:

```
fputs(buffer,fp);
```

9.2.2.3. Formatierte Ein-/Ausgabe

Prototyp:

```
int fscanf(FILE *fp, char *format, ...);
```

Aktion: Vgl. scanf(); die Daten werden lediglich statt von stdin der übergebenen Datei fp entnommen.

Rückgabe: Im Erfolgsfalle liefert fscanf() die Anzahl der ausgelesenen und abgespeicherten Parameter zurück, ansonsten EOF.

Prototyp:

```
int fprintf(FILE *fp, char *format, ...);
```

Aktion: Analog zu printf(); fprintf() schreibt jedoch in die über den FILE-Pointer fp geöffnete Datei statt nach stdout.

Rückgabe: Die Anzahl der geschriebenen Bytes oder einen negativen Wert im Fehlerfall.

9.2.2.4. Blockweise Ein-Ausgabe

Prototyp:

```
size_t fread(void *buf, size_t size, size_t n, FILE *fp);
```

Aktion: fread() liest aus der Datei mit dem FILE-Pointer fp n*size Bytes in den übergebenen Buffer buf ein.

Rückgabe: Die Funktion liefert die Anzahl der erfolgreich gelesenen Einheiten (nicht Bytes) zurück oder 0 im Fehlerfalle.

Beispiel:

```
fread(buf, sizeof(struct Kundendaten), 100, fp);
```

Prototyp:

```
size_t fwrite(void *buf, size_t size, size_t n, FILE *fp);
```

Aktion: Die Funktion fwrite() schreibt in die Datei mit dem FILE-Pointer fp n*size Bytes aus dem Buffer buf.

Rückgabe: Die Funktion liefert die Anzahl der erfolgreich geschriebenen Einheiten (nicht Bytes) zurück oder 0 im Falle eines Fehlers.

Beispiel:

```
fwrite(buf, sizeof(struct Kundendaten), 2000, fp);
```

9.2.3 Schließen der Datei

Prototyp:

```
int fclose(fp);
```

Aktion: fclose() schließt die Datei, auf die fp zeigt.

Rückgabe: 0 im Erfolgsfalle, EOF im Falle eines Fehlers.

9.2.4 Beispiele zur sequentiellen Dateiarbeit

Die nachfolgenden Beispielprogramme files1.c und files2.c sollen die in den vorigen Abschnitten vorgestellten Funktionen im konkreten Programm zeigen.

```

/* files1.c */
#include <stdio.h>
#include <stdlib.h> /* für EXIT_SUCCESS, EXIT_FAILURE */

#define STRLEN 128

int main(void)
{
    FILE *fp;
    char line[STRLEN], filename[STRLEN]="/tmp/probedatei";

    if (fp=fopen(filename,"w")) /* fp != NULL ? */
    {
        fprintf(fp,"Eine Zeile Text...");
        if (fclose(fp)==EOF)
        {
            fprintf(stderr,"\nFehler beim Schließen der Datei "
                "%s!\n",filename);
            return(EXIT_FAILURE);
        }
    }
    else
    {
        fprintf(stderr,"\nFehler beim Öffnen der Datei %s!\n",filename);
        return(EXIT_FAILURE);
    }
    return(EXIT_SUCCESS);
} /* end main */

```

Die Erfahrung zeigt, daß insbesondere die beiden Funktionen fread() und fwrite() für Anfänger in C größere Schwierigkeiten bereiten. Darum hierzu ein konkretes Beispiel, das Programm files2.c.

```

/* files2.c */
#include <stdio.h>
#include <stdlib.h> /* für EXIT_SUCCESS, EXIT_FAILURE */

#define STRLEN 128
#define ZEILEN 4
#define SPALTEN 5

int main(void)
{
    FILE *fp;
    int a[ZEILEN][SPALTEN], i, j, wert=1;
    char filename[STRLEN]="/tmp/probedatei";

    /* Initialisieren des Arrays a mit Kontrolldaten */
    for (i=0; i<ZEILEN; i++)
        for (j=0; j<SPALTEN; j++)
            a[i][j]=wert++;
}

```

```

/* Schreiben in die Datei filename */
if ((fp=fopen(filename,"wb"))==NULL)
{
    fprintf(stderr,"\nFehler beim Öffnen der Datei %s!\n",filename);

    return(EXIT_FAILURE);
} /* end if fopen-Fehler */
fwrite( &a, sizeof(int), ZEILEN*SPALTEN, fp);
if (fclose(fp)==EOF)
{
    fprintf(stderr,"\nFehler beim Schließen der Datei %s!\n",filename);
    return(EXIT_FAILURE);
} /* end if fclose-Fehler */

/* Zur Kontrolle: Lesen der Datei */
/* Davor: Array a mit Nullen belegen */
for (i=0; i<ZEILEN; i++)
    for (j=0; j<SPALTEN; j++)
        a[i][j]=0;
if ((fp=fopen(filename,"rb"))==NULL)
{
    fprintf(stderr,"\nFehler beim Lesen der Datei %s!\n",filename);
    return(EXIT_FAILURE);
} /* end if fopen-Fehler */
fread( a, sizeof(a), 1, fp);
if (fclose(fp)==EOF)
{
    fprintf(stderr,"\nFehler beim Schließen der Datei %s!\n",filename);
    return(EXIT_FAILURE);
} /* end if fclose-Fehler */

/* Kontrollausgabe des Dateiinhalts */
for (i=0; i<ZEILEN; i++)
{
    for (j=0; j<SPALTEN; j++)
        printf("%4d ",a[i][j]);
    printf("\n");
}

return(EXIT_SUCCESS);
} /* end main */

```

Das erwartungsgemäße Ablauflisting dieses Programms sieht so aus:

```

1    2    3    4    5
6    7    8    9   10
11   12   13   14   15

```

9.3. Wahlfreier Zugriff (random access)

Neben den rein sequentiellen Routinen zum Lesen oder Schreiben einer Datei (fscanf(), fprintf()) sieht ANSI C auch einige in stdio.h deklarierte Funktionen für den wahlfreien Zugriff (*random access*) vor.

Mit fseek() kann der File-Pointer auf eine bestimmte Position gesetzt werden, mit rewind() wird der Dateizeiger speziell auf den Anfang der Datei gesetzt, mit ftell() kann abgefragt werden, an welcher Position sich der File-Buffer momentan befindet, mit fread() und fwrite() schließlich können Datensätze (bzw. gleich größere Datenmengen) gelesen und geschrieben werden.

Das nachstehende Programm- und Ablauflisting files3.c soll dies etwas konkreter illustrieren.

```

/* files3.c */
#include <stdio.h>
#include <stdlib.h> /* für EXIT_SUCCESS, EXIT_FAILURE */
#include <string.h>

typedef struct Kunde
{
    int kundenr;
    char nachname[30];
    char vorname[25];
} KUNDE;

int main(void)
{
    FILE *fp;
    char *filename="/tmp/probe";
    KUNDE kunde, kunde2, kunde3;

    /* Zur Demonstration: Initialisierung von kunde */
    strcpy(kunde.nachname,"$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$");
    strcpy(kunde.vorname,"!!!!!!!!!!!!!!!!!!!!!!!!!!!!");
    kunde.kundenr=255;

    /* Öffnen der Datei filename zum Schreiben, ggf. Anhängen */
    fp=fopen(filename,"wb+");
    if (fp==NULL)
    {
        fprintf(stderr,"\nFehler: Datei %s kann nicht "
            "geöffnet werden!\n",filename);
        return(EXIT_FAILURE);
    } /* end if fp==NULL */

    /* Schreiben von drei Datensätzen in die Datei filename */
    strcpy(kunde.nachname,"Asimov");

```

```

strcpy(kunde.vorname,"Isaak");
kunde.kundenr=10;
fwrite(&kunde,sizeof(KUNDE),1,fp);
strcpy(kunde.nachname,"Böll");
strcpy(kunde.vorname,"Heinrich");
kunde.kundenr=11;
fwrite(&kunde,sizeof(KUNDE),1,fp);
strcpy(kunde.nachname,"Canetti");
strcpy(kunde.vorname,"Elias");
kunde.kundenr=12;
fwrite(&kunde,sizeof(KUNDE),1,fp);

/* "Zurückspulen" an den Anfang der Datei */
rewind(fp);
/* äquivalent zu fseek(fp,0,SEEK_SET); */

/* Lesen des ersten Datensatzes */
fread(&kunde2,sizeof(KUNDE),1,fp);
printf("Der erste Datensatz: ");
printf("%4d: %s %s\n",
    kunde2.kundenr,kunde2.vorname,kunde2.nachname);

/* Lesen des dritten Datensatzes (ohne Fehlerbehandlung) */
fseek(fp,(3-1)*sizeof(KUNDE),SEEK_SET); /* Auf 3.Position */
fread(&kunde3,sizeof(KUNDE),1,fp);
printf("Der dritte Datensatz: ");
printf("%4d: %s %s\n",
    kunde3.kundenr,kunde3.vorname,kunde3.nachname);

/* Ändern des zweiten Datensatzes */
fseek(fp,1L*sizeof(KUNDE),SEEK_SET); /* Auf 2.Position */
fread(&kunde,sizeof(KUNDE),1,fp);
strcpy(kunde.nachname,"Blum");
strcpy(kunde.vorname,"Katharina");
fseek(fp,1L*sizeof(KUNDE),SEEK_SET); /* Auf 2.Position */
fwrite(&kunde,sizeof(KUNDE),1,fp);

/* Anhängen eines vierten Datensatzes */
fseek(fp,0,SEEK_END); /* an die letzte Position */
/* fseek(fp,3L*sizeof(KUNDE),SEEK_SET); /* Auf 2.Position */
strcpy(kunde.nachname,"Dürrenmatt");
strcpy(kunde.vorname,"Friedrich");
kunde.kundenr=14;
fwrite(&kunde,sizeof(KUNDE),1,fp);

/* Lesen aller Datensätze der Datei filename */
rewind(fp);
printf("\nDie Datensätze in %s sind nun:\n",filename);
do
{

```

```

    if (fread(&kunde,sizeof(KUNDE),1,fp)==1)
        printf("%4d: %s %s\n",
            kunde.kundennr,kunde.vorname,kunde.nachname);
} while (!feof(fp));

/* Programm beenden, return code EXIT_SUCCESS */
fclose(fp);
return(EXIT_SUCCESS);

} /* end main */

```

Nachstehend noch das Ablauflisting von files3.c sowie, vielleicht ganz interessant, ein Blick in die Datendatei /tmp/probe, so wie sie auf einem PC unter DOS von dem Programm angelegt wird. (Zusatzfrage für Kenner: woran könnten Sie der Datei ansehen, daß sie auf einem PC erstellt worden ist?)

Ablauflisting:

```

Der erste Datensatz:    10: Isaak Asimov
Der dritte Datensatz:  12: Elias Canetti

```

Die Datensätze in /tmp/probe sind nun:

```

10: Isaak Asimov
11: Katharina Blum
12: Elias Canetti
14: Friedrich Dürrenmatt

```

Inhalt der Datei tmp/probe:

hexadezimal:	ASCII:
0A 00 41 73 69 6D 6F 76 00 24 24 24 24 24 24 24	..Asimov.\$\$\$\$\$\$
24 24 24 24 24 24 24 24 24 24 24 24 24 24 00	\$\$\$\$\$\$\$\$\$\$\$\$\$.
49 73 61 61 6B 00 21 21 21 21 21 21 21 21 21	Isaak.!!!!!!!!
21 21 21 21 21 21 21 21 00 2E 0B 00 42 6C 75 6D	!!!!!!!...Blum
00 76 00 24 24 24 24 24 24 24 24 24 24 24 24	.v.\$\$\$\$\$\$\$\$\$\$\$\$
24 24 24 24 24 24 24 24 24 00 4B 61 74 68 61 72	\$\$\$\$\$\$\$\$\$.Kathar
69 6E 61 00 21 21 21 21 21 21 21 21 21 21 21	ina.!!!!!!!!
21 21 00 2E 0C 00 43 61 6E 65 74 74 69 00 24 24	!!....Canetti.\$\$
24 24 24 24 24 24 24 24 24 24 24 24 24 24 24	\$\$\$\$\$\$\$\$\$\$\$\$\$\$\$
24 24 24 00 45 6C 69 61 73 00 63 68 00 21 21 21	\$\$\$.Elias.ch.!!!
21 21 21 21 21 21 21 21 21 21 21 00 2E 0D 00	!!!!!!!.....
44 81 72 72 65 6E 6D 61 74 74 00 24 24 24 24 24	Dürrenmatt.\$\$\$\$
24 24 24 24 24 24 24 24 24 24 24 00 46 72	\$\$\$\$\$\$\$\$\$\$\$\$\$.Fr
69 65 64 72 69 63 68 00 21 21 21 21 21 21 21	iedrich.!!!!!!!!
21 21 21 21 21 21 00 2E	!!!!!!..

10 Ergänzungen

In diesem Kapitel sollen einige weiterführende Themen angeschnitten werden, die in der Praxis der C-Programmierung eine wichtige Rolle spielen. Dem derzeitigen Trend zum PC hin folgend werden einige dieser Ergänzungen auf der Grundlage von DOS und Windows, andere auf der originären C-Plattform UNIX vorgestellt werden. Die entsprechenden Tools (Werkzeuge) existieren jedoch so oder in sehr ähnlicher Form auch auf den jeweils anderen Betriebssystemen.

10.1. make und Makefiles

Die Entwicklung vor allem größerer Softwareprojekte wird unter UNIX und von den meisten PC-Entwicklungssystemen mit dem make-Konzept unterstützt. Alternativ dazu bietet z.B. Borland bei seinem Turbo C sogenannte Projectfiles an, die jedoch – lediglich intern anders abgespeichert – einen ähnlichen Ansatz verfolgen.

Das elementare make-Kommando von UNIX wurde bereits angesprochen. Mit

```
make beispiel
```

wird aus dem Sourcefile `beispiel.c` das Programm `beispiel` (statt `a.out`) erstellt. In dieser Form entspricht `make beispiel` also dem Aufruf

```
cc -o beispiel beispiel.c
```

`make` erkennt dabei sogar, ob eine Compilation evtl. nicht erforderlich ist, weil das ausführbare Programm jünger als der C-Quelltext ist! Näheres zu `make` kann unter UNIX online über *man make* abgerufen werden.

Komplizierter ist die Situation jedoch bei Multi-File-Projekten, also Programmen, deren Code in mehreren Dateien abgelegt ist. Hier helfen sogenannte Makefiles, die bei UNIX standardmäßig auch Makefile oder makefile heißen. In diesen werden die verschiedenen Abhängigkeiten abgespeichert, so daß das `make`-Kommando gezielt genau und nur die gerade erforderlichen Neu-Compilationen veranlassen muß. Bei diesem Verfahren spricht man von inkrementellem Compilieren (und Linken), weil nur das erneut compiliert bzw. gebunden wird, was nicht bereits auf dem neuesten Stand vorhanden ist.

Das UNIX-Kommando `mkmf` (*make makefile*) erstellt (zu allen `.c` und `.h` Dateien in einem Verzeichnis ein entsprechendes Makefile; dies soll anhand des nachstehenden Mini-Projektes illustriert werden.

Gehen wir aus von der folgenden Datei `main.c`, die ein kleines Hauptprogramm beinhaltet.

```

/* main.c */
#include <stdio.h>
#include "main.h"

int a[MAXIMUM][MAXIMUM];

void main(void)
{
    InitArray(a);
    PrintArray(a);
} /* end main */

```

In diesem Hauptprogramm werden zwei Funktionen aufgerufen, die nicht in dieser Quelldatei zu finden sind; deklariert werden sie in der Headerdatei main.h, die nachstehend gezeigt wird.

Das Vorgehen, mit `#ifndef` abzufragen, ob eine symbolische Konstante bereits definiert worden ist, ist typisch für Headerdateien: damit soll verhindert werden, daß ein und dasselbe Headerfile in einem Projekt (versehentlich) mehrfach eingebunden wird, wo es nicht erforderlich oder eventuell auch gar nicht zulässig ist.

```

/* main.h */
#ifndef MAIN_H__
#define MAIN_H__

#define MAXIMUM (10)

void InitArray(int a[MAXIMUM][MAXIMUM]);
void PrintArray(int a[MAXIMUM][MAXIMUM]);

#endif /* MAIN_H__ */

```

Die Dateien `init.c` und `print.c` beinhalten jeweils den Quelltext für die Funktionen `InitArray()` bzw. `PrintArray()`, die in `main.c` verwendet werden.

```

/* init.c */
#include "main.h"

void InitArray(int a[MAXIMUM][MAXIMUM])
{
    int i, j;
    for (i=0; i<MAXIMUM; i++)
        for (j=0; j<MAXIMUM; j++)
            a[i][j]=i*MAXIMUM+j;
} /* end InitArray */

```

```

/* print.c */
#include "main.h"
#include <stdio.h>

void PrintArray(int a[MAXIMUM][MAXIMUM])
{
    int i, j;
    for (i=0; i<MAXIMUM; i++)
    {
        for (j=0; j<MAXIMUM; j++)
            printf("%4d ",a[i][j]);
        printf("\n");
    }
    printf("\n");
} /* end PrintArray */

```

Wird in einem Verzeichnis, in dem sich diese vier Dateien (main.c, main.h, init.c, print.c) befinden, das Kommando mkmf abgesetzt, so entsteht (hier nur gekürzt und vereinfacht wiedergegeben) folgendes Makefile. Der Backslash \ dient innerhalb des Makefiles (wie bei C) als Fortsetzungszeichen; das Zeichen # ist bei Makefiles der Beginn eines Zeilenkommentars.

```

# Makefile
# Dieses Makefile wurde mit mkmf automatisch erzeugt!
HDRS      = main.h
LD        = cc
OBJS      = init.o \
           main.o \
           print.o
PROGRAM   = main
SRCS      = init.c \
           main.c \
           print.c

all:      $(PROGRAM)

$(PROGRAM): $(OBJS) $(LIBS)
           @echo "Linking $(PROGRAM) ..."
           @$ (LD) $(LDFLAGS) $(OBJS) $(LIBS) -o $(PROGRAM)
           @echo "done"

clean:;   @rm -f $(OBJS) core

###
init.o:   main.h
main.o:   /usr/include/stdio.h main.h
print.o:  /usr/include/stdio.h main.h

```

Und so arbeitet make mit dem Makefile zusammen: wird make (oder hier synonym dazu: make all) aufgerufen, so wird nachgesehen, ob es das ausführbare Programm main bereits gibt; wenn ja, dann wird überprüft, ob es jünger als alle beteiligten .o-Dateien (Objectfiles) ist. Wenn nein, dann werden die Abhängigkeiten zwischen den .o-Dateien und den im Makefile genannten Headerfiles und (implizit) den gleichnamigen .c-Dateien überprüft.

Im nachstehenden Beispiel sind zunächst main und die Objectfiles gar nicht vorhanden, so daß make alles neu erstellen muß. Es werden also sämtliche .c-Dateien compiliert, dann sämtliche .o-Dateien zusammengebunden zur ausführbaren Programmdatei main.

Ein Ablaufprotokoll mit make unter UNIX³¹:

```
$ mkmf /* Makefile wird erstellt */
$ make

        cc -O -c init.c
        cc -O -c main.c
        cc -O -c print.c
Linking main ...
done
$ touch init.c
/* touch tut so, als ob init.c verändert worden wäre, */
/* konkret wird das Dateizugriffsdatum aktualisiert, */
/* make auf diese Weise also "betrogen". */
$ make
        cc -O -c init.c
Linking main ...
done
$ touch main.h
$ make all
        cc -O -c init.c
        cc -O -c main.c
        cc -O -c print.c
Linking main ...
done
$ make all
Target `main' is up to date. Stop.
$
```

³¹Das Dollarzeichen \$ ist der übliche Prompter (die Eingabeaufforderung) der Bourne- oder Korn-Shell unter UNIX.

10.2. Debugger

Ein Debugger³² ist ein Programm, das einem Software-Entwickler hilft, den eigenen Code auf Fehler zu untersuchen. Dabei ermöglicht ein guter Debugger es, sich den Code auf Assemblerebene und – im Falle von symbolischen Debuggern – im Quelltext (z.B. in C) anzusehen und ihn schrittweise ablaufen zu lassen. Dabei kann jede Anweisung einzeln abgearbeitet werden, es kann aber auch nach jedem Funktionsaufruf oder an sonst zu definierenden Stellen, sogenannten *Breakpoints*, angehalten werden. Der Debugger zeigt dabei wahlweise auch die aktuellen Registerinhalte, Funktionsaufrufe, Stack- und Variablenbelegungen an.

Solche Debugger gibt es praktisch für jede Betriebssystemplattform; die Arbeitsweise ist im wesentlichen immer die gleiche: man vermutet einen Fehler in einer Funktion *f()* und sieht sich daher den Ablauf dieser Funktion genauer an, zum Beispiel im Einzelschrittverfahren. Dabei betrachtet man auch die jeweils aktuellen Werte der Variablen, insbesondere eventueller Pointer³³.

Betrachten wir etwa das folgende kleine Programm *test1.c*: wir sehen recht schnell, daß hier mit *a[4]* ein ungültiger Array-Zugriff erfolgt, denn das deklarierte Array besteht lediglich aus den Komponenten *a[0]* bis *a[3]*. Dies wird jedoch kein normaler C-Compiler als Fehler melden, denn hinter der Indexschreibweise steckt bekanntlich die Zeigerarithmetik.

```
/* test1.c - Demo zum Debugger */
void main(void)
{
    int dummy=99, a[4], i;
    for (i=1; i<=4; i++)
        { /* Fehlerhafter Indexzugriff! */
            a[i]=i-4;
            printf("a[%d]=%5d\n",i,a[i]);
        }
} /* end main */
/* end of file test1.c */
```

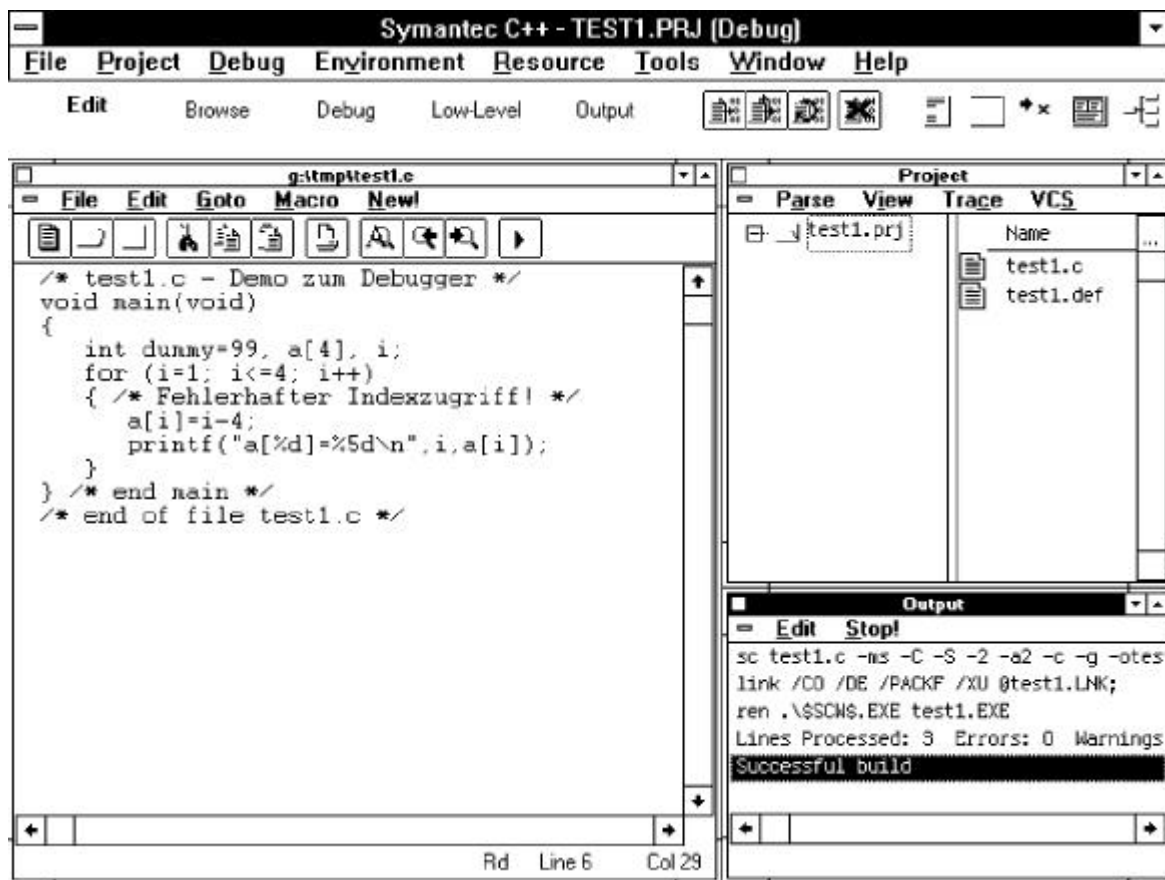
In einem Debugger können wir uns nun die schrittweise Abarbeitung dieses kleinen Programms sowie die Belegung der verschiedenen Variablen-Speicherplätze ansehen. Insbesondere können wir somit nachvollziehen, wieso dieses kleine Programm in eine Endlosschleife gerät!³⁴

³²engl. bug - die Wanze, to debug - entwanzen, im Rahmen der Software-Entwicklung: den Programmcode von Fehlern befreien

³³Gerade in Pointern enthaltene ungültige Adressen sind eine der Hauptfehlerquellen größerer Programme, die mit dynamischer Speicherplatzallokierung arbeiten.

³⁴Diese Endlosschleife ist nicht zwingend, – dabei handelt es sich vielmehr um eine compilerabhängige Verhaltensweise. Übungsaufgabe für die Profis: woran liegt es, daß dieses Programm in der hier gezeigten Fassung in eine Endlosschleife gerät?

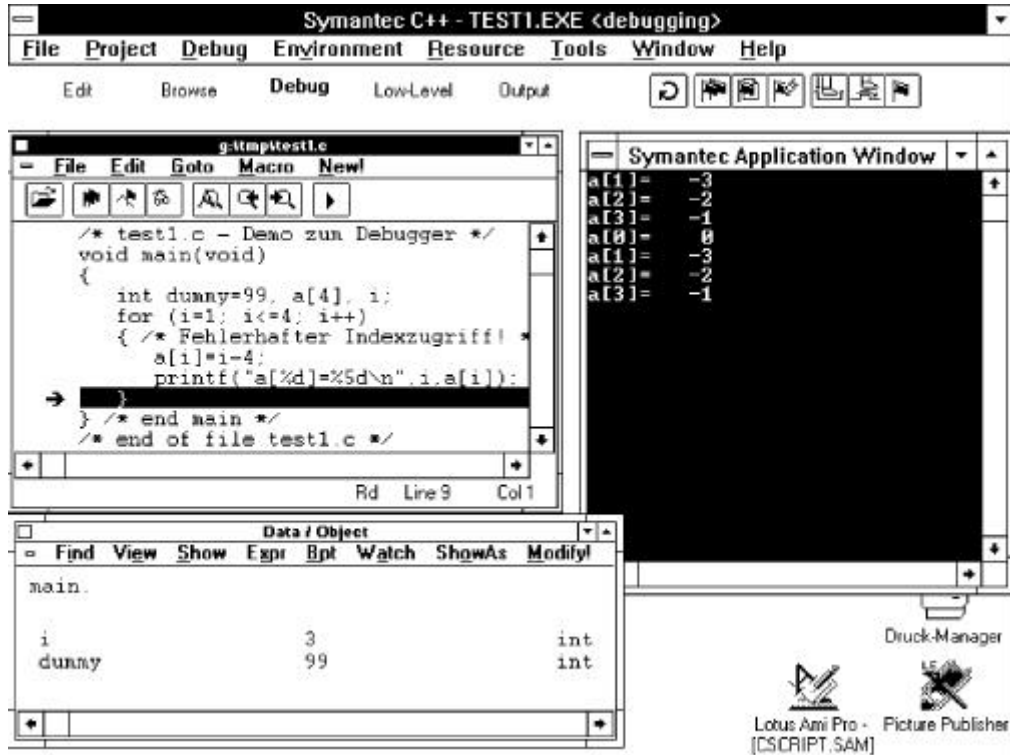
Beispielhaft für die Vielzahl von Debuggern, die es auf dem Markt gibt, werden hier zwei Schnappschüsse des Symantec C++ 7.0 Debuggers (für Windows 3.1/95 und Windows NT) gezeigt.



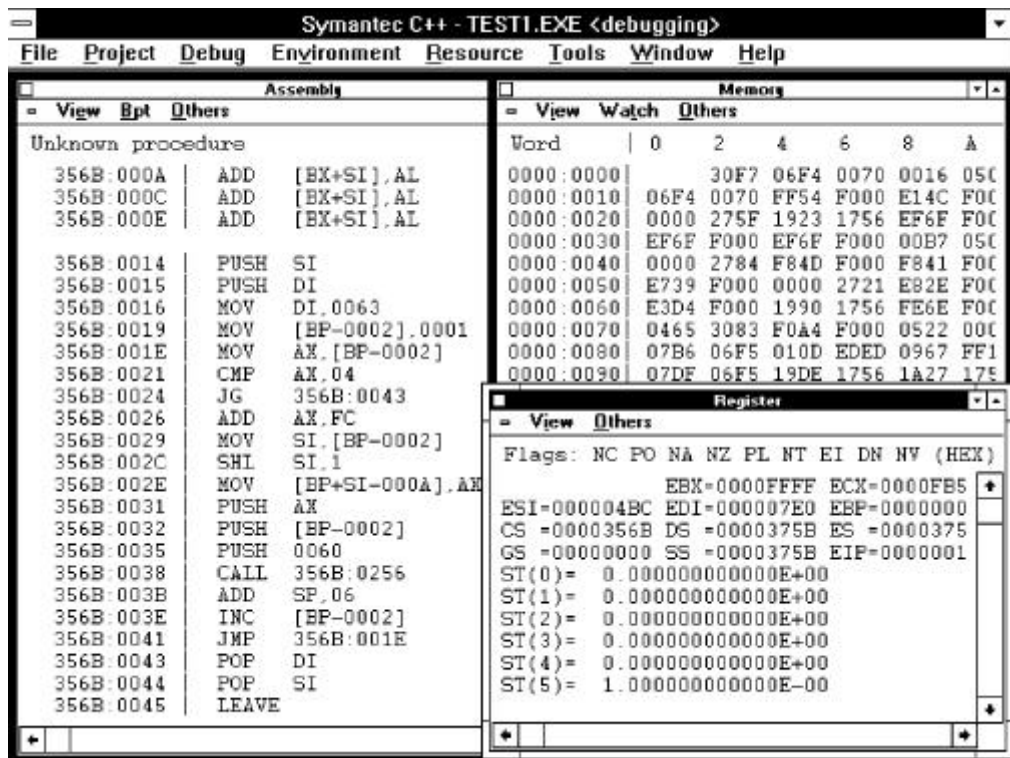
Im obigen Bild sehen wir die Entwicklungsumgebung des Symantec C++ Compilers³⁵. Das Projekt TEST1.PRJ, bestehend aus der einen Quelltextdatei TEST1.C sowie – windows-typisch – einer Definitionsdatei TEST1.DEF, ist im Debug-Modus geladen, d.h. zu der ausführbaren Datei werden Debugger-Informationen hinzugefügt, die das symbolische Debuggen – also mit Referenz auf den C-Quelltext – ermöglichen.

Im nachstehenden Bildschirmschnappschuß sieht man das Quelltextfenster, in dem ein dicker Pfeil auf die aktuelle Quelltextposition verweist; daneben ist das "Application Window", in dem das ausführbare Programm – hier ein textbasiertes DOS-Programm – abläuft. Darunter werden in dem Fenster "Data/Object" die aktuellen Variablenbelegungen angezeigt.

³⁵Die Firma Symantec hat in den frühen Neunziger Jahren die Firma Zortech übernommen, die einen der besten und traditionsreichsten C++ Compiler auf den Markt gebracht hat. Selbstverständlich liefern auch die anderen Compilerhersteller wie Borland, Microsoft oder Watcom entsprechende Debugger zu ihren Entwicklungssystemen an.



Wie in dem Anwendungsfenster zu sehen ist: das Programm liefert nicht die erwarteten vier Ausgabezeilen für `a[1]` bis – fehlerhafterweise – `a[4]`, sondern nach `a[1]` bis `a[3]` wird plötzlich `a[0]` gezeigt! Der Debugger hilft hier bei der Fehlersuche indem er zeigt, daß beim Index `i=4` "`a[4]`" (d.h. `*(a+4)`) auf 0 gesetzt wird; und die Adresse `a+4` verweist (bei diesem Compiler) gerade auf den Speicherplatz `i`!



Daneben können, wie das obige Bild zeigt, auch der Assemblercode, die Speicher- oder Registerinhalte angesehen werden³⁶.

10.3. Profiler

Unter einem Profiler versteht man ein Dienstprogramm, welches die Performance (Leistungswerte) eines Programmes analysiert, indem es feststellt, wie lange welche Quellcode-Zeilen oder Module aktiv sind, wie oft Module oder einzelne Anweisungen aufgerufen werden (und von wem) oder auf welche Dateien wie oft und für wie lange von dem Programm zugegriffen wird. Gleichzeitig können weitere Aktivitäten überwacht bzw. protokolliert werden, z.B. Druckausgaben, Systemaufrufe, Tastatureingaben. Zur Ermittlung eventuell ineffizienter Programmteile dient der Profiler ähnlich wie ein Debugger bei der Fehlersuche.

Im folgenden soll aus Übersichtlichkeitsgründen nur ein ganz kleines Beispiel eines C-Programms mit Borland's PC-Turbo Profiler ansatzweise analysiert werden. Ohne die graphische Aufbereitung stehen unter UNIX ebenfalls Profiler zur Verfügung; mit man gprof kann der Manual-Eintrag zum Profiler gprof aufgelistet werden. Dieser korrespondiert mit der C-Compileroption -G, der eine Programmdatei für die Analyse mit gprof erstellt.

Zunächst der Quelltext des kleinen Demonstrationsprogramms, in dem exemplarisch untersucht werden soll, ob eine for-Schleife der Art

```
for (i=0; i<strlen(s); i++)
```

besonders ineffizient ist, da die Funktion strlen() hier bei jedem Schleifendurchlauf erneut aufgerufen wird³⁷.

```
/* profctest.c
 * Kleines Demonstrationsprogramm für den Turbo Profiler von Borland.
 */
#include <stdio.h>
#include <string.h>

void main(void)
{
    char s[] = "Programmieren in C";
    int i, sl;

    for (i=0; i<strlen(s); i++)
        putchar(s[i]);

    sl=strlen(s);
    for (i=0; i<sl; i++)
        putchar(s[i]);
}
```

³⁶Solche sehr hardwarenahen Aspekte werden in der Regel hexadezimal, also im Sechzehnersystem, dargestellt.

³⁷Dies gilt jedenfalls solange der Compiler diese Konstruktion nicht optimiert.

Das entsprechend (in diesem Fall mit Borlands Turbo C) compilierte EXE-Programm wird nun in den Profiler geladen.

Der Profiler kann nun eine Statistik über einen oder mehrere Programmläufe (im gezeigten Bild sind es zehn) erstellen. Im nachstehend gezeigten Bild sehen wir einen Bildschirmabzug des Turbo Profiler von Borland.

```

File View Run Statistics Print Options Window Help
Module: PROFTEST File: PROFTEST.C 4
#include <stdio.h>
#include <string.h>

-> void main(void)
  (
    char s[] = "Programmieren in C";
    int i, sl;

->   for (i=0; i<strlen(s); i++)
->     putchar(s[i]);
  )

[1]=Execution Profile
Total time: 0.5449 sec      Display: Time and counts
% of total: 78 %         Filter: All
Runs: 10 of 10           Sort: Frequency

#PROFTEST#10      180 43%
#PROFTEST#14      180 43%
#PROFTEST#14      0.1775 sec 41%
#PROFTEST#10      0.2519 sec 58%

```

Im oberen Teil ist der Quellcode zu sehen, im unteren das sogenannte *Execution Profile*, das Ausführungsprofil. Hierbei wird gezeigt, wie oft (hier: 180 mal) und wie lange jeweils einzelne Anweisungen durchlaufen wurden. Hier wurde das `putchar()` in der Schleifenformulierung `for (i=0; i<strlen(s); i++)` 180mal aufgerufen, dafür wurden 0,2519 Sekunden benötigt. Demgegenüber haben die `putchar()`-Aufrufe in der Schleifenkonstruktion `for (i=0; i<sl; i++)` bei gleicher Aufrufzahl nur 0,1775 Sekunden gebraucht.

Dieses einfache Beispiel zeigt somit schon recht gut, wie ein Profiler bei der statistischen Analyse und dem Aufspüren von Engpässen (*bottle necks*) behilflich sein kann.

```

File View Run Statistics Print Options Window Help
Module: PROFTEST File: PROFTEST.C 4
#include <stdio.h>
#include <string.h>

=> void main(void)
  (
    char s[] = "Programmieren in C";
    int i, sl;

=>   for (i=0; i<strlen(s); i++)
=>     putchar(s[i]);
  )

[1]=CPU 80486
main: void main(void)
=>7381:0239 55      push    bp
7381:023A 8BEC      mov     bp,sp
7381:023C 83EC18    sub     sp,0018
7381:023F 39269A00  cmp    [009A],sp
7381:0243 7203     jb     0248
7381:0245 E8910B    call   F_OVERFLOW
7381:0248 16       push   ss
7381:0249 8D46E8    lea   ax,[bp-18]
7381:024C 50       push  ax

```

Daneben erlauben Profiler in der Regel noch eine Reihe weiterer Untersuchungsmethoden, beispielsweise können die Aufruf-Hierarchien (welche Funktion ruft welche auf?) oder – etwas mehr low-level orientiert – wie im obigen Bild gezeigt der generierte Assemblercode angezeigt werden.

10.4. Cross-Reference-Listen mit cxref

Ein weiteres Hilfsmittel bei komplexeren Programmierungen sind sogenannte Cross-Reference-Listen, bei denen textlich oder graphisch dargestellt wird, welche Funktionen welche anderen Funktionen (in welchen Modulen) aufrufen und welche Konstanten (aus welchen Headerfiles) sie verwenden.

Nachstehend exemplarisch die Online-Hilfe zum UNIX-Kommando `cxref`, das eine solche Cross-Reference-Liste erzeugt. Danach folgt ein kleines Beispielprogramm (mit nur einem Modul) und die von `cxref` dazu produzierte – hier allerdings stark verkürzt wiedergegebene – Cross-Reference-Liste.

NAME

`cxref` - generate C program cross-reference

SYNOPSIS

`cxref` [options] files

DESCRIPTION

`cxref` analyzes a collection of C files and attempts to build a cross-reference table. `cxref` utilizes a special version of `cpp` to include `#defined` information in its symbol table. It produces a listing on standard output of all symbols (auto, static, and global) in each file separately, or with the `-c` option, in combination. Each symbol contains an asterisk (*) before the declaring reference. Output is sorted in ascending collation order (see Environment Variables below).

Options

In addition to the `-D`, `-I`, and `-U` options (which are identical to their interpretation by `cc` (see `cc(1)`), `cxref` recognizes the following options:

- `-c` Print a combined cross-reference of all input files.
- `-w num` Width option; format output no wider than `num` (decimal) columns. This option defaults to 80 if `num` is not specified or is less than 51.
- `-o file` Direct output to the named file.
- `-s` Operate silently; do not print input file names.
- `-t` Format listing for 80-column width.

- Aa Choose ANSI mode. If not specified, compatibility mode (-Ac option) is selected by default.
- Ac Choose compatibility mode. This option is selected by default if neither -Aa nor -Ac is specified.

EXTERNAL INFLUENCES

Environment Variables

LCCOLLATE determines the order in which the output is sorted.

If LCCOLLATE is not specified in the environment or is set to the empty string, the value of LANG is used as a default. If LANG is not specified or is set to the empty string, a default of ``C'' (see lang(5)) is used instead of LANG. If any internationalization variable contains an invalid setting, cxref behaves as if all internationalization variables are set to ``C'' (see environ(5)).

International Code Set Support

Single- and multi-byte character code sets are supported with the exception that multi-byte character file names are not supported.

DIAGNOSTICS

Error messages are unusually cryptic, but usually mean that you cannot compile these files anyway.

EXAMPLES

Create a combined cross-reference of the files orange.c, blue.c, and color.h:

```
cxref -c orange.c blue.c color.h
```

Create a combined cross-reference of the files orange.c, blue.c, and color.h: and direct the output to the file rainbow.x:

```
cxref -c -o rainbow.x orange.c blue.c color.h
```

WARNINGS

cxref considers a formal argument in a #define macro definition to be a declaration of that symbol. For example, a program that #includes ctype.h will contain many declarations of the variable c.

cxref uses a special version of the C compiler front end. This means that a file that will not compile probably cannot be successfully processed by cxref. In addition, cxref generates references only for those source lines that are actually compiled. This means that lines that are excluded by #ifdefs and the like (see cpp(1)) are not cross-referenced.

cxref does not parse the CCOPTS environment variable.

FILES

/lib/cpp	C-preprocessor.
/lib/xpass	Compatibility-mode special version of C compiler front end.
/lib/xpass.ansi	ANSI-mode special version of C compiler front end.

STANDARDS CONFORMANCE

cxref: SVID2, XPG2, XPG3

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

void CallMe1(void);
void CallMe2(void);

void main(void)
{
    CallMe1();
} /* end main */

void CallMe1(void)
{
    CallMe2();
    CallMe2();
} /* end CallMe1 */

void CallMe2(void)
{
    printf("Hello, world!\n");
} /* end CallMe2 */
```

Cross-Reference-Liste zu cxrefdemo.c

cxrefdemo.c (stark gekürzt):

SYMBOL	FILE	FUNCTION	LINE
BUFSIZ	/usr/include/stdio.h	--	*16
CallMe1()	cxrefdemo.c	--	*3 11
	cxrefdemo.c	main	8
CallMe2()	cxrefdemo.c	--	*4 17
	cxrefdemo.c	CallMe1	13 14
EOF	/usr/include/stdio.h	--	96 *97
stdin	/usr/include/stdio.h	--	*100
stdout	/usr/include/stdio.h	--	*101

11 Literaturhinweise

An dieser Stelle sollen einige wenige Literaturhinweise gegeben werden.

Cuber, U. und Wenzel, H.

Das Einmaleins der C-Programmierung.

Düsseldorf, 1994.

Dem Buch liegt der MS-DOS-C/C++-Kommandozeilencompiler von Symantec in der Version 6.11 auf CD-ROM bei.

Hatton, L.

Safer C.

Berkshire, 1995.

Ein englischsprachiges Buch für etwas Fortgeschrittene, in dem es um Möglichkeiten geht, die C-Programmierung etwas sicherer zu gestalten.

Kelley, A. und Pohl, I.

C - Grundlagen und Anwendungen.

Bonn, 1987.

Kernighan, B. und Ritchie, D.

The C Programming Language (2nd edition).

Englewood Cliffs, 1988.

Das Standardwerk.

Kernighan, B. und Ritchie, D.

Programmieren in C.

(Deutsche Version des o.g. Buches.)

München, 1990.

Das Standardwerk in deutscher Sprache. Die 2.Auflage behandelt dann ANSI-C.

Küster, H.-G.

Umsteigen von COBOL auf C und C++

Vaterstetten/München, 1993.

Es soll ja noch COBOL-Programmierer geben. Diese finden hier einen Weg zurück in das Leben...

Müldner und Steele

C as a Second Language for Native Speakers of Pascal.

Reading, 1988.

Wer artig Pascal als erste Programmiersprache erlernt hat, ist mit diesem Buch gut beraten.

Plum, T.

Learning to Program in C.

Cardiff, 1983.

Stevens, W. R.

Advanced Programming in the UNIX Environment.

Reading, 1992.

Wie der Titel es sagt: speziell für die tiefergehende Programmierung unter dem Betriebssystem UNIX ist dieses Buch empfehlenswert.

Tondo, C. L. und Gimpel, S. E.

Das C-Lösungsbuch zu Kernighan & Ritchie.

München, 1987.

Ward, R.

Debugging C.

Bonn, 1988.

12 Index

@!/?#<:>_-=

!, 19
&, 20
,, 21
<, 19
=, 20
>, 19
^, 20
|, 20
~, 20
!=, 19
%=:, 20
&&, 19
&=:, 20
*=:, 20
+=:, 20
-=:, 20
/=, 20
<<=:, 20
<=:, 19
==, 19
>=:, 19
>>=:, 20
^=:, 20
|=, 20
||, 19
<<=:, 20
>>=:, 20
?!, 25
??, 25
?!(, 25
?)), 25
?)-, 25
?)/, 25
?<=:, 25
?>=:, 25
?>=:, 25
\\0, 32
#define, 23
#elif, 25
#else, 25
#endif, 25
.h, 23
#if, 25
#ifdef, 25
#ifndef, 25
#include, 6, 23
/usr/include, 29

A

Abbruchbedingung, 59
Adresse, 47
all purpose language, 6
Alternative, 37
ANSI, 5
ANSI-C, 5
argc, 56
Argument, 56
argument counter, 56
argument vector, 57
argv, 56
Arithmetische Operatoren, 18
Array, 46
Arrays von Pointern, 54
ASCII, 13, 15
ASCII-Zeichen Nr. 0, 32
Assembler, 80
assert.h, 29
Assoziativität, 22
auto, 9, 44
automatic, 6

B

B, 6
babylonische Keilschrift, 6
Backslash, 15
Backspace, 15
Baum, 63
BCPL, 6
bedingte Compilation, 25
Bedingungsoperator, 21
Bell, 15
Bewertungsreihenfolge, 22
Bibliothek, 6
binär, 18
Binärdatei, 68
binärer Baum, 63
Bit-Felder, 51
bit fields, 51
blaue Wunder, 32
Blockstruktur, 43
boolean, 19
Borland, 50, 83
break, 8, 40
Breakpoint, 80

C

C++, 6, 18, 26
call by reference, 26
call by value, 26
Carriage Return, 15
case, 8
cast-Operator, 19
Casting, 17

char, 9
const, 10, 45
continue, 8, 40, 41
Cross-Reference-Liste, 85
ctype.h, 29
cxref, 85

D

Datei, 67
Dateizeiger, 67
Debugger, 80
default, 8
div, 18
do, 8
do-while-Schleife, 38
double, 9
Dreizeichenfolge, 24
dynamisch, 52

E

else, 8
entry, 8
enum, 9, 16
environment, 56
environment pointer, 57
envp, 57
ermo.h, 29
exit, 40
exit(), 41
EXIT_FAILURE, 42
EXIT_SUCCESS, 42
extern, 9, 44
external linkage, 44
externe Bindung, 43, 44
externe Einflüsse, 45

F

Fakultät, 59
fclose(), 68, 70
Fehlerausgabekanal, 30
Fehlerkanal, 67
Feld, 46
fgetc(), 68, 69
fgets(), 68
file, 67
File-Pointer, 67
Flag, 40, 51
float, 9
float.h, 29
fopen(), 68
for, 8
for-Schleife, 38
FormFeed, 15
fprintf(), 68, 70
fputc(), 68, 69

fputs(), 68, 69
Fragezeichenoperator, 21
fread(), 68, 70, 71, 73
Freiheit, 54
fscanf(), 68, 70
fseek(), 73
ftell(), 73
Funktion, 26
Funktionsaufruf, 80
fußgesteuert, 38
fwrite(), 68, 70, 71, 73

G

Ganzzahldivision, 18
getchar(), 30
Gleitkommatyp, 14
goto, 8, 42
gprof, 83
Gültigkeitsbereich, 44

H

Hardware-Register, 44
Hauptprogramm, 7
hexadezimal, 15
high byte, 50

I

if, 8
if-Anweisung, 37
if-else-Anweisung, 37
INCLUDE, 47
Include-File, 23
Indexbereich, 47
Inkrementoperator, 18
int, 9
INT_MAX, 14
integer-Datentyp, 14
isalpha(), 31
isdigit(), 31
isirgendwas(), 31
islower(), 31
Iteration, 38

K

K&R-C, 28
Kernighan, 5
Kernighan-Ritchie-C, 28
Klingelzeichen, 15
Kommandozeilenargument, 56
Komponente, 48
Konstante, 15
Kontrollstruktur, 37
kopfgesteuert, 38

L

Lebensdauer, 44
lex, 5
Library, 6
limits.h, 29
lineare Liste, 60
LineFeed, 15
lint, 5
locale.h, 29
long, 9
long int, 9
low byte, 50

M

main(), 7
make, 76
make makefile, 76
Makefile, 76
Makros mit Parametern, 23
malloc(), 60
math.h, 29
MAXINT, 14
Mehrfachverzweigung, 40
Menge, 46
Microsoft, 81
mkmf, 76
mod, 18
Modul, 43
Modulo-Operator, 18

N

Nachfolger, 63
Negation, 19
NewLine, 15

O

Object code, 29
ODER, 19
oktal, 15
old fashioned style, 28
Optimierung, 45
ordinal, 14

P

Parameter, 28
Pascal, 6
Performance, 83
Pointer, 52
Pointer auf Array, 54
Pointerarithmetik, 52
Preprocessor, 6
printf, 8
printf(), 33
Priorität, 22

Profiler, 83
Prototyp, 27
putchar(), 31

Q

Quelltext, 43

R

random access, 73
Referenz (C++), 26
Referenzparameter (C++), 26
Referenzübergabe, 26
Register, 50
register, 9, 44
Registerinhalt, 80
Rekursion, 26, 59
return, 8, 40
rewind(), 73
Ritchie, 5

S

scanf(), 27, 33, 35
SCCS, 5
Schleife, 38
Seitenvorschub, 15
sequentielle Dateiverarbeitung, 68
Sequenzoperator, 21
set, 46
setjmp.h, 29
short, 9
signal.h, 29
signed, 9, 13
signed int, 9
sizeof, 8, 13
Softwareprojekt, 26, 76
Sourcefile, 43
Spaß, 7
Speicheradresse, 52
Speicherklasse, 44
Stack, 80
Standard-Headerfile, 29
standard library, 29
Standardausgabe, 67
Standardbibliothek, 29
Standardeingabekanal, 67
static, 9, 44
stdarg.h, 29
stddef.h, 29
stderr, 30, 67
stdin, 30, 67
stdio.h, 29
stdlib.h, 29
stdout, 30, 67
strcpy(), 32
string.h, 29

Stroustrup, 6
struct, 9, 48
Struktur, 48
strukturierte Programmierung, 37, 42
switch, 8, 40
switch-Anweisung, 40
Symantec, 81, 89

T

Tabulatorzeichen, 15
ternär, 21
Textdatei, 68
time.h, 29
tolower(), 31
toupper(), 31
Trigraph, 24
Turbo C, 50
Turbo Profiler, 83
typedef, 9, 16, 48
Typkonversion, 17

U

Umgebungsbereich, 56
Umgebungsinformation, 57
UND, 19
ungültige Adresse, 80
union, 9, 49
UNIX, 5, 29
unsigned, 9, 13

V

Variablenbelegung, 80
Variante Struktur, 49
Verbund, 48
Vergleichsoperator, 19
Verschachtelung von Funktionen, 43
void, 9, 16
volatile, 10, 45
Vorgänger, 63

W

Wagenrücklauf, 15
wahlfreier Zugriff, 73
Watcom, 81
Wertübergabe, 26
while, 8
while-Schleife, 38
Wurzel, 63

X

XOR, 20

Y

yacc, 5

Z

Zählschleife, 38
Zeichenkette, 31
Zeiger, 52
Zeiger auf Zeiger, 54
Zeilenvorschub, 15
Zortech, 81
Zuweisungsoperator, 20
zweidimensionales Array, 47
zweistellig, 18

Dieses Dokument [ansi-c.lwp] wurde
erstellt mit Lotus Word Pro. © 1995-1999
P.Baemle, Bergisch Gladbach
Revision 3.2 vom 28.04.99
Insgesamt 93 Seiten.