Hartwig

Von BASIC zu C mit dem ATARI® ST

EIN DATA BECKER BUCH

Hartwig

Von BASIC zu C mit dem ATARI® ST

EIN DATA BECKER BUCH

ISBN 3-89011-147-5

Wichtiger Hinweis:
Die in diesem Buch wiedergegebenen Schaltungen, Verfahren und Programme werden ohne Rücksicht auf die Patentlage mitgeteilt. Sie sind ausschließlich für Amateur- und Lehrzwecke bestimmt und dürfen nicht gewerblich genutzt werden.

Alle Schaltungen, technischen Angaben und Programme in diesem Buch wurden von dem Autoren mit größter Sorgfalt erarbeitet bzw. zusammengestellt und unter Einschaltung wirksamer Kontrollmaßnahmen reproduziert. Trotzdem sind Fehler nicht ganz auszuschließen. DATA BECKER sieht sich deshalb gezwungen, darauf hinzuweisen, daß weder eine Garantie noch die juristische Verantwortung oder irgendeine Haftung für Folgen, die auf fehlerhafte Angaben zurückgehen, übernommen werden kann. Für die Mitteilung eventueller Fehler ist der Autor jederzeit dankbar.

Inhaltsverzeichnis

TEIL 3: WEITERFÜHRENDE SPRACHELEMENTE

TEIL 4: NACHSCHLAGETEIL

ANHANG

Einleitung

"Von BASIC zu 'C' auf dem ATARI ST" ist in seiner Konzeption bisher einzigartig. Es wurde ganz speziell für BASIC-Programmierer erstellt, die auf 'C' "umsteigen" oder besser "aufsteigen" und damit u.a. die fantastischen Möglichkeiten des ATARI ST wie beispielsweise GEM und die Graphik nutzen wollen.

Es gibt vielfältige Gründe, um als BASIC-Programmierer 'C' zu erlernen. Diese moderne Sprache besitzt eine sehr große **Flexibilität**. Die **Ausführungsgeschwindigkeit** entspricht etwa der von reinem Maschinencode, ist also ca. 100mal so schnell als die des ST-BASICs.

Ein weiterer entscheidender Punkt ist die **Portabilität**, die Übertragbarkeit von 'C'-Software. Einmal erstellt, können Sie Programme, die aus nur aus den standardisierten 'C'-Grundsprachelementen bestehen, auf jedem System ausführen, für das ein 'C'-Compiler erhältlich ist. Denken Sie hier einmal zurück an die Unzahl nicht kompatibler, maschinenspezifischer BASIC-Dialekte.

Einer der Hauptgründe, 'C' zu erlernen, liegt jedoch in der neuen Computergeneration: Der Superrechner ATARI ST bietet mit seinem 68000-Prozessor so viel Prozessorleistung, daß eine Assemblerprogrammierung unsinnig wird. BASIC ist ebenfalls "out".

Gefragt ist hier 'C'. Alle professionellen Programme auf dem ST werden in 'C' geschrieben. Viele Möglichkeiten dieses Computers wie das GEM lassen sich auch nur mittels 'C' erschließen. Ohne Zweifel ist es für Sie von unschätzbarem Vorteil, wenn Sie 'C' so schnell wie möglich beherrschen lernen.

Jedoch haben BASIC-Programmierer oft die meisten Probleme, den Schritt von BASIC zu 'C' zu machen. Dies liegt daran, daß 'C' bisher die Programmiersprache einer Elite von Software-

Designern und Systemprogrammierern auf leistungsfähigen Computern war, BASIC dagegen die typische und verbreitete Einsteigersprache bei Home-Computer Anwendern darstellte.

Diese krasse Diskrepanz können Sie durch das vorliegende Buch überwinden. Wollten Sie bisher 'C' erlernen, so waren Sie auf herkömmliche 'C'-Bücher angewiesen, die sich ausschließlich an den absoluten Programmieranfänger wenden. Außerdem waren sie rechnerunabhängig und gehen damit auf die **speziellen Möglichkeiten des ST** nicht ein.

Viele der dargestellten Informationen stellen jedoch für Sie mit Ihren BASIC-Erfahrungen lediglich unnötigen Ballast dar. Andererseits tauchen durch Ihre BASIC-Kenntnisse ganz spezifische Probleme sowie typische Fehler auf, die oft zu ungeahnten Schwierigkeiten mit 'C' führen können.

Dieses 'C'-Buch, das speziell auf den ATARI ST abgestimmt wurde, begeht einen völlig neuen Weg, indem diese Schwierigkeiten berücksichtigt und gelöst werden:

Jedes **'C'-Sprachelement** wird in **'C'-Programmen** veranschaulicht. Zum Vergleich erhalten Sie immer das entsprechende **BASIC-Programm**. Sie können so den Zusammenhang zwischen BASIC und 'C' sofort erfassen. Alle Gemeinsamkeiten sowie die Unterschiede zu BASIC werden detailliert erörtert.

"Von BASIC zu 'C' auf dem ATARI ST" gliedert sich in vier große Teile:

Im einleitenden Abschnitt erhalten Sie einen kurzen Überblick über den Stellenwert sowie die Entwicklung und Anwendung von 'C'.

Dann folgt eine **direkte Einführung in 'C'**. Speziell für BASIC-Programmierer werden die elementaren Grundelemente erörtert. Alle unnötigen Detailinformationen, Sonderfälle und Einschränkungen sind ausgelassen.

Das Ziel dabei ist, Sie so schnell wie möglich in die Lage zu versetzen, eigene 'C'-Programme zu erstellen. Das Kapitel ist so konzipiert, daß Sie voraussichtlich bereits nach *einem einzigen Tag* mit den 'C'-Grundstrukturen selbständig Ihre ersten eigenen, einfachen Programme erstellen können.

Auf diesen Grundinformationen baut der zweite große Abschnitt auf. In diesem wenden wir uns detailliert den **Teilaspekten, Anwendungen** und wichtigen **Sonderfällen** der Basiselemente von 'C' zu.

Aspekte wie Bildschirmoperationen, Variablen, Zeiger, arithmetische Ausdrücke sowie die Kontrollstrukturen werden jeweils anschaulich an BASIC-Beispielen erörtert.

Den Abschluß dieses Teils bildet ein Kapitel, in dem die charakteristischen und spezifischen **'C'-Fehler von BASIC-Umsteigern** vorgestellt und erklärt werden.

Mit dem so gefestigten Basiswissen wenden wir uns im dritten Teil des Buches den **weiterführenden Sprachelementen** zu.

Wieder im anschaulichen Vergleich mit BASIC werden **Funktionen** (Unterprogramme), **Strukturen** (Verbundvarialblen) und die **Pointer** detailliert beschrieben.

Der vierte Teil bildet einen umfassenden **'C'-Überblick**. Speziell für BASIC-Umsteiger werden die für die Programmierpraxis entscheidenden 'C'-Sprachelemente zum schnellen Nachschlagen übersichtlich dargestellt.

Mit Hilfe dieser Konzeption können Sie also schnell und effektiv **"um"**- und **"aufsteigen"** und die neuen Möglichkeiten von 'C' auf Ihrem ATARI ST erschließen.

Ich wünsche Ihnen dabei viel Spaß und Erfolg!

Olaf Hartwig

TEIL 1:
DER EINSTIEG IN 'C'

Kapitel 1:

Entwicklung, Anwendung und Stellenwert von 'C' speziell auf dem ATARI ST

'C' ist eine universelle Programmiersprache für allgemeine Anwendungen. Sie ist eine relativ neue Sprache, die von Dennis Ritchie und den Bell Laboratories (AT&T) entwickelt wurde.

Unter "high-level"-Software-Designern gilt 'C' als die **Programmiersprache der Zukunft.** Dies liegt vor allem an ihrem **kompakten Sprachumfang,** dem sehr ökonomischen Ausdrucksmöglichkeiten und der Fähigkeit zum maschinennahen Programmieren.

Man kann 'C' auch als "Kurzschriftversion" von Assembler betrachten, da 'C' **reinen Maschinencode** generiert. Viele 'C'-Programmierer waren ehemalige Maschinencode-Programmierer.

'C' ist sehr maschinennah. Das bedeutet, daß 'C' mit den gleichen Objekten umgeht wie der Computer, mit Zeichen, Zahlen und Speicheradressen.

Vor allem dieser Eigenschaft verdankt 'C' seinen Einsatz in der professionellen Softwareentwicklung. Nahezu alle hochleistungsfähigen Softwarepakete, wie beispielsweise das universelle, integrierte Tabellenkalkulationsprogramm "Lotus 1-2-3" oder "Lotus Synphony" genauso wie das neue "Jazz" sind 'C'-Programme.

Auch alle Programme für den ATARI ST sowie das gesamte ATARI-Betriebssystem und GEM sind ausschließlich in 'C' geschrieben.

'C' ist eine sehr kompakte Sprache. Sie besitzt etwa genauso viele Befehle wie BASIC. Daher sollte es für Sie auch nicht weiter schwierig sein, sich diese Befehle und deren Syntax recht schnell anzueignen.

Die große Stärke von 'C' liegt in den vielfältigen Funktionen, die in Bibliotheken abgelegt sind.

Der Anwender kann Funktionen für die verschiedensten Anwendungen erstellen. Durch den freien Zugriff auf die **Bibliotheken** ist es für den Programmierer unnötig, bestimmte Funktionen immer wieder neu zu erstellen.

Die Funktionen sind auch der Grund für die unglaubliche **Flexibilität** dieser Sprache. Vor allem ist 'C' damit auf die Zukunft ausgerichtet. So ist beispielsweise das moderne GEM des ST nichts anderes als eine Erweiterung von 'C', da es über eine Reihe von Bibliotheken direkt über zahlreiche neue Funktionen adressierbar ist.
Da 'C' jedoch sehr maschinennah ist, also auf einer niedrigeren Ebene als beispielsweise Cobol, Pascal oder BASIC steht, haben viele Programmierer Schwierigkeiten, auf 'C' umzusteigen.

Dies gilt ganz besonders für BASIC-Programmierer. Sehr schwierig wird der Wechsel von BASIC zu 'C', wenn Sie bereits eine sehr große BASIC-Praxiserfahrung besitzen. Da fast alle nicht professionellen Programmierer BASIC als erste Programmiersprache lernen, mag dies ein Hauptgrund dafür sein, warum bisher fast kein BASIC-Programmierer den Wechsel zu 'C' wagte.

Aber gerade auf dem ATARI ST sollten Sie als ehemaliger BASIC-Programmierer so schnell wie möglich auf 'C' umsteigen. Obwohl das ST-BASIC sehr ausgefeilt ist, ermöglicht es nicht die Erstellung fortgeschrittener Programme.

Der Hauptgrund dafür liegt darin, daß GEM über BASIC überhaupt nicht ansprechbar ist. Und gerade die fantastischen Mög-

lichkeiten von GEM und der Maussteuerung sind neben der Prozessorgeschwindigkeit und dem Mega-RAM des 520 ST PLUS die herrausragenden Möglichkeiten dieses Computers.

Wollen Sie beispielsweise die **Graphikmöglichkeiten** des ST auch nur ansatzweise nutzen, haben Sie hier nur die Möglichkeit, entweder in **Maschinensprache** oder aber in **'C'** zu programmieren. Das ST-BASIC eröffnet mit seinen wenigen Graphik-Befehlen nur einen minimalen Bruchteil der fantastischen Graphik auf dem ST.

In Maschinensprache zu programmieren wird auf dem ST unsinnig. Bei Geschwindigkeitsmessungen zwischen dem direkt eingegebenen Maschinencode und dem reinen Maschinencode, der von dem 'C'-Compiler generiert wurde, ließ sich in fast allen Fällen kein meßbarer Geschwindigkeitsunterschied festgestellen.

Der von einem 'C'-Compiler erzeugte reine Maschinencode ist so effektiv, daß er sich oft auch in Assembler nicht mehr optimieren läßt.

Gleichzeitig ist das Programmieren in Maschinencode etwa 6 bis 10mal so zeitaufwendig wie das Programmieren in dem strukturierten 'C'. Maschinenprogramme lassen sich ferner nur sehr schwer modifizieren, in 'C' sind beliebige Änderungen dagegen sagenhaft einfach.

Ferner lassen sich Programme in Maschinencode nicht auf Computer mit anderen Prozessoren übertragen. Selbst bei gleichen Prozessoren wird eine Umwandlung zur Qual für jeden Programmierer. Dagegen lassen sich die auf dem ST programmierten 'C'-Programme problemlos auf alle anderen Computer wie den AMIGA oder den IBM PC übertragen.

Damit sind Ihre heute auf dem ST erstellten Programme auch **zukunftssicher**. Wenn Sie später von dem ST auf ein noch moderneres System umsteigen, können Sie Ihre 'C'-Programme sofort übernehmen und somit auch in Zukunft anwenden.

Wie leistungsstark 'C' ist, zeigt sich ja auch daran, daß nicht nur das superschnelle GEM sondern auch das alle von DIGITAL geschriebenen Routinen wie LOAD, SAVE, etc. nicht im Maschinencode, sondern in 'C' geschrieben wurden.

Mit diesem Buch haben Sie erstmals die Möglichkeit, von BASIC zu 'C' aufzusteigen. Da es ausgesprochen typische Schwierigkeiten sind, die BASIC-Umsteiger in 'C' blockieren, konnte auf all diese gesondert eingegangen werden. Zudem ist das gesamte Konzept des Buches speziell für BASIC-Programmierer entwickelt und gezielt auf die Möglichkeiten des ST abgestimmt worden.

Kapitel 2:

Erste Schritte für BASIC-Umsteiger

2.1. ERLERNEN SIE DIE 'C'-GRUNDSTRUKTUREN AN EINEM TAG!

Sie haben richtig gelesen. Für die Durcharbeitung dieses Kapitels werden Sie voraussichtlich nur *einen einzigen Tag* benötigen.

Trotzdem stelle ich Ihnen hier alle wesentlichen Strukturen von 'C' vor. Anhand dieser sind Sie in der Lage, nach der Beendigung dieses Kapitels Ihre ersten 'C'-Programme auf Ihrem ST selbständig zu entwickeln.

Ihnen wird hier also ein schneller und einfacher Einstieg in 'C' ermöglicht. Dabei konzentrieren wir uns auf die **wichtigsten Sprachelemente**, ohne uns in Details, Einschränkungen, Sonderfällen und Regeln zu verlieren.

Dieses Kapitel erhebt also keinerlei Anspruch, vollständig und umfassend zu sein.

Die Details werden dann zusammen mit vielen weiteren Ausführungen sowie Tips und Tricks zum effektiven Programmieren in 'C' in den folgenden Kapiteln dieses Buches erörtert.

Ich möchte Sie mit dieser knappen, aber präzisen Einführung in diesem Kapitel in kürzester Zeit dahin bringen, die ersten eigenen 'C'-Programme zu erstellen.

Die Konzeption hat natürlich auch **Nachteile**. Abgesehen von der Unvollständigkeit der Ausführungen werden wir später bei der Detailvorstellung von 'C' nicht umhin kommen, Teile dieses

Kapitels zu wiederholen. Ich glaube jedoch, daß sich diese Wiederholungen nicht störend, sondern vielmehr lernfördernd auswirken werden.

Eine neue Programmiersprache lernt man nur, wenn man in ihr experimentiert und Programme schreibt.

Daher steigen Sie nun ein in 'C', um in kürzester Zeit den Punkt zu erreichen, an dem Sie eigenständig Ihre eigenen Programme schreiben können.

2.2. DIE FUNKTIONEN UND TEXTAUSGABE AUF DEM BILDSCHIM

Ein elementares Problem einer jeden Programmiersprache ist die Ausgabe von Kommentaren und Meldungen auf dem Bildschirm.

Wie erreichen Sie beispielsweise, daß der Computer den Kommentar

```
Hallo wie gehts?
```

auf dem Display ausgibt?

In BASIC geschieht dies natürlich durch den einfachen PRINT-Befehl:

```
10 PRINT "HALLO WIE GEHTS?"
```

In 'C' sieht das entsprechende Programm ähnlich aus, nämlich:

```
main()
{

      printf("Hallo wie gehts?\n");
      gemdos(0x1);

}
```

Auf dem DIGITAL-'C' des ATARI ST ist zusätzlich zu der Textausgabe über die Funktion **'printf()'** der GEMDOS-Aufruf

```
gemdos(0x1);
```

an das Programm anzufügen.

Dieser bewirkt, daß die Programmausführungen solange angehalten werden, bis eine beliebige Taste gedrückt wird. Auch bei den folgenden Beispielen müssen Sie diese GEM-spezifische Besonderheit berücksichtigen und **an das Ende der Programme jeweils den GEMDOS-Aufruf anhängen.**

Noch ein Wort zur Compilierung der einzelnen 'C'-Programme: Bedingt durch die gewaltigen Möglichkeiten des ATARI ST gestaltet sich der Compilierungsvorgang oft recht aufwendig.

So müssen z.B. unterschiedliche LINK-Prozesse durchgeführt werden, je nachdem, ob das jeweilige 'C'-Programm eine **TOS**-Anwendung, eine **GEM**-Applikation oder eine **DESK**-Accessory darstellt.

Aufgrund der vielfachen Möglichkeiten und der oft komplexen Compilierungsprozesse werden die populärsten 'C'-Compiler auf dem ATARI ST in dem Data Becker Buch "TIPS & TRICKS ZU 'C' AUF DEM ATARI ST" gesondert behandelt. Mehr über diesen Band erfahren Sie im Anhang dieses Buches.

An unserem Beispielprogramm

```
main()
{

    printf("Hallo wie gehts?\n");
    gemdos(0x1);

}
```

fällt sofort die Eröffnung durch

```
    main()
```

auf.

Dieser Befehl stellt eine besondere **'Funktion'** von 'C' dar. Zur Erklärung: 'C'-Programme bestehen normalerweise aus einer Folge einzelner Funktionen.

Diese Funktionen sind in etwa mit den Subroutinen von BASIC vergleichbar, die durch GOSUB aufgerufen und mit RETURN beendet werden. Sie entsprechen auch den Pascal-Programmierern bekannten Prozeduren.

Jeder Funktion wird in 'C' ein Name zugewiesen, z.B.:

```
eingabe()
{
     Hier stehen dann die Befehle
     der Funktion Eingabe.
}
```

'main()' ist jedoch eine besondere Funktion. Sie wird bei der Ausführung des endgültigen Programms immer als erste aufgerufen und ausgeführt. Damit stellt sie einen Steuerungskopf des 'C'-Programms dar.

Ganz wichtig ist hier, festzuhalten, daß jedes 'C'-Programm die Funktion 'main()' irgendwo enthalten muß. Diese ruft dann normalerweise andere Funktionen, also weitere Unterprogramme auf.

Die in unserem einfachen Beispielprogramm auf den Programmkopf folgenden geschweiften Klammern umgeben alle Anweisungen, aus denen sich eine Funktion zusammensetzt.

Die letzte Klammer läßt sich mit dem END-Befehl des BASIC vergleichen.

Kommen wir nun zu der Zeile

```
printf("Hallo wie gehts?\n");
```

Um uns das Leben einfach zu machen, setzen wir

```
printf(      );
```

vorläufig erst einmal mit dem

```
PRINT-Befehl
```

des BASIC gleich, obwohl **"printf"** wesentlich mehr leistet und auch von der Syntax her komplizierter ist als der BASIC-Befehl PRINT. Auszugebender Text muß wie in unserem Beispiel durch die runden Klammern sowie die Anführungszeichen eingeschlossen werden.

Nun zur Bedeutung des Zeichens '\n' indem Programmtext: Dieses Zeichen wird nicht ausgegeben, sondern stellt einen Zeilenende-Marker dar. Es ist dabei für die Ausführung des **Zeilenvorschubs** verantwortlich.

Hätten Sie diesen Marker ausgelassen, also die Zeile so eingegeben

```
printf("Hallo wie gehts?");
```

so entspräche dies der BASIC-Anweisung

```
10 PRINT "Hallo wie gehts" ;
```

Das Semikolon würde hier einen Zeilenvorschub unterbinden.

2.3 DAS PROGRAMMFORMAT

Sie könnten unser Programm auch so in 'C' formulieren:

```
main()
{

        printf("Hallo *");
        printf("wie ");
        printf("gehts?");
        printf(" n");
        gemdos(0x1);

}
```

Dies ist auch in BASIC möglich. Das Programm lautet dann folgendermaßen:

```
10 PRINT "Hallo ";
20 PRINT "wie ";
30 PRINT "gehts?";
40 PRINT
```

An dieser Stelle möchte ich Sie noch auf einen wesentlichen **Unterschied** zwischen dem 'C'-Compiler und dem ST-BASIC-Interpreter aufmerksam machen:

Der 'C'-Compiler akzeptiert ein beliebiges Programmformat. Das bedeutet, daß Sie unser Programm auch folgendermaßen formulieren könnten:

```
main()
{

     printf("Hallo
     wie
     gehts n");
     gemdos(0x1);

}
```

Normalerweise nimmt der Compiler auch diese Eingabe an, ein BASIC-Interpreter dagegen nicht. Sie sollten sich jedoch **nicht** einen derartigen Programmierstil angewöhnen, da die Lesbarkeit Ihrer Programme sehr darunter leidet. Einige Compiler, wie beispielsweise das ST-DIGITAL-'C' akzeptieren eine derartige Eingabe auch nicht. Das DIGITAL 'C' gibt einen

```
string cannot cross line
```

Error aus.

Beachten Sie einmal die **Semikolons** in unserem 'C'-Programm. Ihre Aufgabe besteht darin, jeweils einzelne Befehle zu trennen, vergleichbar mit den Doppelpunkten in BASIC oder auch dem Trennungssemikolon in PASCAL.

2.4. NUMERISCHE BILDSCHIRMAUSGABEN

Wir haben nun erste Erfahrungen mit dem Programmformat gemacht, uns mit Funktionen beschäftigt und sind in der Lage, Texte auf dem Bildschirm auszugeben.

Die entsprechende **Ausgabe von Zahlen** ist jedoch nicht ganz so einfach.

In BASIC ist auch dies, genau wie bei der Ausgabe von Texten, über einen einfachen PRINT-Befehl möglich. Nehmen wir einmal das folgende BASIC-Programm als Beispiel:

```
10 A=1
20 B=3.14
30 PRINT A;B
40 END
```

Das entsprechende 'C'-Programm lautet wie folgt:

```
main()
{

    int a;
    float b;

       a = 1;
       b = 3.14;

    printf("%7d  %5.1f\n", a, b);
    gemdos(0x1);

}
```

2.4.1. DIE VARIABLENDEKLARATIONEN

Zunächst einmal benötigen wir noch einige grundsätzliche Erklärungen zur Struktur einer Funktion in 'C':

Alle Variablen innerhalb dieser Routine müssen eingangs **deklariert** werden. In BASIC ist das normalerweise nicht nötig.

Die Deklaration teilt dem Computer mit, welche Variablen er verwenden soll, definiert also die **Variablennamen.** In diesem Fall sind dies a und b. Außerdem werden die **Variablentypen** festgelegt:

```
int a;
```

Die Variable "a" stellt dabei eine ganzzahlige **Integer- Zahl** dar, festgelegt durch den Variablentyp **"int"**.

Die Deklaration

```
float b;
```

legt "b" als **Gleitkommavariable** fest.

Zwischen diesen beiden Variablentypen muß man sehr genau unterscheiden - eine Differenzierung, die in BASIC nicht vorgenommen wird.

2.4.2. DIE INITIALISIERUNG VON VARIABLEN

Nach der Deklaration muß als nächstes eine **Wertezuweisung** erfolgen. In unserem Beispielprogramm sieht das folgendermaßen aus:

```
a = 1;
b = 3.14;
```

Auch dies wird Ihnen als BASIC-Umsteiger ungewohnt und umständlich vorkommen. Wenn Sie bisher ein BASIC-Programm erstellt hatten, dann wurden durch RUN alle Variablen auf Null gesetzt.

In 'C' geschieht dies jedoch **nicht**. Auch die vorangegange Deklaration weist den Variablen noch keine Werte, auch keine Null, zu.

Die Wertezuweisung müssen Sie immer **gesondert** durchführen. So unnötig Ihnen das als BASIC-Programmierer erscheinen mag, nach einiger Erfahrung in 'C' werden Sie feststellen, daß Ihre Programme dadurch wesentlich strukturierter, besser nachvollziehbar und veränderbar werden.

2.4.3. DIE FORMAT-ANWEISUNGEN

Kommen wir nun zu der 'C'-Programmzeile:

```
printf("%7d   %5.1f\n",a ,b);
```

Im Vergleich zu der BASIC-Anweisung

```
PRINT A; B
```

mutet sie anfangs reichlich kompliziert an. Doch kann sie auch wesentlich mehr leisten, wie Sie gleich sehen werden.

Die Grundstruktur des "printf"-Befehls kennen Sie ja bereits aus den Beispielen des Textausdrucks.

Neu sind lediglich die **Steuerungsanweisungen** innerhalb der Anführungszeichen. Dabei handelt es sich um sogenannte Format-Anweisungen.
Ein Beispiel dafür ist das Format:

```
"%5.1f"
```

Jede Format-Anweisung beginnt mit einer "%"-Einführung. Diese teilt dem Computer mit, daß er einen Wert, in diesem Fall die Variable "b", in einem bestimmten Ausgabeformat anzeigen soll.

Die **Formatbestimmung** wird durch das Element **"5.1"** festgelegt. Die "5.1" bedeutet dabei, daß die Zahl in einer Fläche dargestellt werden soll, die fünf Zeichen umfaßt und eine Nachkommastelle beinhaltet.

Der Index **"f"**, der das **Umwandlungs-Element** darstellt, teilt dem Computer mit, daß es sich um die Ausgabe einer Gleitkommazahl, eine **"float"**-Variable handelt. Entsprechend steht das Element **"d"** für eine **Integervariable**, einen ganzzahligen Wert.

Die Format-Anweisung ist äußerst effektiv. Der Aufwand einer Ausgabenformatierung ist in BASIC erheblich größer und umständlicher als in 'C' über den "printf"-Befehl.

2.5 SCHLEIFEN UND KOMMENTARE

Nur mit Ausgaben von Variablen oder Texten können wir jedoch nicht sinnvoll programmieren.

Wir benötigen noch Schleifen, entsprechend beispielsweise der FOR-NEXT-Wiederholungsschleife in BASIC.

2.5.1 DIE "FOR"-SCHLEIFE

Hier haben Sie es als BASIC-Umsteiger leicht, die Schleifenkonstruktion zu verstehen, da sie sehr mit der **FOR-NEXT** Schleife verwandt ist. Dies werden Sie sofort am folgenden Beispielprogramm sehen. Es erstellt eine Liste einer quadratischen Funktion mit Werten von eins bis zwanzig:

```
main()
{

    int x;

    for (x = 1; x <= 20; x = x + 1)
        printf("%2d  %3d\n", x, x*x);

    gemdos(0x1);

}
```

Der Programmlauf sieht folgendermaßen aus:

```
1       1
2       4
3       9
4       16
5       25
6       36
7       49
8       64
9       81
10      100
.
.
.
18      324
19      361
20      400
```

Und hier kommt das entsprechende BASIC-Programm zum anschaulichen Vergleich mit der "for"-Schleife in 'C':

```
10  FOR X=1 TO 20 STEP +1
20        PRINT X; X*X
30  NEXT X
40 END
```

Dieses BASIC-Programm ist absolut identisch mit dem vorherigen 'C'-Programm. Die Programmstruktur des 'C'-Programms ist uns ja bereits bekannt. Zunächst wurde die Funktion "main()" eröffnet, anschließend mußte die Variable "x" als Integer, also ganzzahlig deklariert werden.

Dann folgte die eigentliche "for"-Schleife:

```
for (x = 1 ; x <=20 ; x = x + 1 )
    printf("%2d %3d\n", x, x*x);
```

Die 'C'-FOR-Schleife läßt sich direkt mit der BASIC-FOR-NEXT-Schleife vergleichen. Zunächst wird der erste x-Wert mit Eins festgelegt.

Aus

```
FOR X=1
```

wird in 'C' die Zuweisung

```
x = 1;
```

Beachten Sie dabei, daß es nicht mehr nötig ist, eine gesonderte Wertezuweisung der Variablen x vorher durchzuführen. Sie müssen also vor der "for"-Schleife nicht zusätzlich 'x = 1;' zur Initialisierung im Programm eingeben.

Die Wertezuweisung wird statt dessen direkt in der Schleife vorgenommen.

Der zweite Teil der 'C'-"for"-Schleife stellt die Bedingung der äußeren Intervallgrenze dar.

Der BASIC-FOR-NEXT-Ausdruck

```
TO 20
```

wird in 'C' mit

```
x <= 20;
```

festgelegt.

Hier sind natürlich auch viele andere Be-grenzungsfestlegungen möglich. Ein Beispiel wäre

```
x < 21;
```

was die gleiche Wirkung wie die Anweisung "x <= 20" hätte.

Der dritte Teil der "for"-Schleife ist die Schrittweite. In BASIC ist es nicht nötig, eine Schrittweite von "+1" zu formulieren.

Wenn Sie eine FOR-NEXT-Schleife ohne die STEP-Anweisung festlegen, setzt der BASIC-Interpreter die Schrittweite automatisch auf "+1".

Bei 'C' müssen Sie dagegen etwas umdenken. Die Schrittweite muß immer festgelegt werden, was in diesem Fall mit

```
x = x + 1
```

geschieht, vergleichbar in BASIC mit der Anweisung

```
STEP +1
```

Beachten Sie bitte auch, daß nach der Schleifenfestlegung, also nach

```
for(erste Intervallgrenze;
                    zweite Grenze;
                              Schrittweite)
```

kein Semikolon folgen darf.

Sicher ist Ihnen aufgefallen, daß ich immer von einer "for"-Schleife gesprochen habe im Vergleich zu einer BASIC-FOR-NEXT-Schleife.

Wo ist das NEXT indem 'C'-Programm geblieben? Oder anders gefragt, wie erkennt der 'C'-Compiler, wie viele Befehle er innerhalb der "for"-Schleife wiederholen soll?

Hier gilt die allgemeine Grundregel: Der Befehl, der auf die Schleifenfestlegung **direkt** folgt, wird so lange wiederholt, wie in dieser Festlegung bestimmt wurde.

In unserem Programm wird die "printf"-Anweisung also zwanzigmal wiederholt.

Wollen Sie jedoch **mehrere Befehle** in die Schleife eingliedern, diese also mehrfach wiederholen, so müssen sie alle in geschweifte Klammern gesetzt werden. Ein entsprechendes Beispiel-Programm sähe folgendermaßen aus:

```
main()
{

   int x;

     for (x = 1 ; x <= 20 ; x = x + 1)
         {
             printf("X-Wert ist...");
             printf("%2d\n", x);
             printf("X-Quadrat ...");
             printf("%3d\n", x*x);
         }
     gemdos(0x1);

}
```

Hier folgt der "for"-Anweisung ein ganzer Block von "printf"-Befehlen, die alle innerhalb der geschweiften Klammern stehen müssen.

Dieses Programm gibt die x-Werte zusammen mit den x-Quadrat-Werten, jeweils durch eine Leerzeile getrennt, untereinander aus:

```
X-Wert ist... 1
X-Quadrat ... 2

X-Wert ist... 2
X-Quadrat ... 4

bis

X-Wert ist... 20
X-Quadrat ... 400
```

2.5.2 DIE "WHILE"-SCHLEIFE

Diese Schleife ist Ihnen als BASIC-Programmierer wohl weniger bzw. bisher nicht bekannt. **While** läßt sich für unseren Zweck am besten mit **"solange"** übersetzen.

Der Algorithmus der Schleife angewendet auf unser vorheriges Beispielprogramm ist folgender:

```
x = 1 ;

solange x <= 20 tue folgendes:
{
      x = x + 1;
      Ausdruck von x
      und von x-Quadrat
}
```

Sie können diesen "while"-Algorithmus auch in BASIC formulieren, was Sie unbewußt sicher auch schon öfters getan haben. Dies sähe wie folgt aus:

```
10 X=1
20 :
30 IF X<=20 THEN PRINT X;X*X:GOTO30
40 :
50 END
```

Das entsprechend formulierte 'C'-Programm mit der korrekten Syntax der "while"-Schleife sehen Sie hier:

```
/* Wertetabelle für X-Quadrat */
main()
{

  int x;
  x = 1 ;

    while (x <= 20)
    {
          printf("X-Wert   ...");
          printf("%2d\n", x);
          printf("X-Quadrat...");
          printf("%3d\n", x*x);

          x = x + 1;
    }
    gemdos(0x1);

}
```

Dieses Programm entspricht exakt unserem letzten Beispiel einer Wertetabelle mittels einer "for"-Schleife.

Hier müssen Sie jedoch nach der Variablendeklaration von "x" als Integer zusätzlich eine Wertezuweisung vornehmen, was in unserem Programm durch

```
x = 1;
```

erfolgt. Dies ist notwendig, damit x einen Ausgangswert erhält. In der "for"-Schleife wurde das innerhalb der Schleifenanweisung abgehandelt.

Die "while"-Syntax ist aus dem Programm klar ersichtlich: Die runden Klammern nach dem "while"-Ausdruck beinhalten die Bedingung, unter der die folgenden Befehle ausgeführt werden sollen.

Die zu wiederholenden Ausdrücke stehen in geschweiften Klammern. Dies entspricht der Syntax der bereits beschriebenen "for"-Schleife.

Soll beispielsweise nur eine Anweisung wiederholt werden, so können die Klammern entfallen.

Auch parallel zur "for"-Schleifen-Syntax folgt nach der "while"-Bedingung kein Semikolon.

Die **Schrittweite** muß innerhalb der Schleifenanweisung bestimmt werden, in diesem Fall durch

```
x = x + 1;
```

Sie können hier auch jede andere Schrittweite wie "2" oder "0.5" angeben.

2.5.3 KOMMENTARE IN 'C'

Nun fehlt lediglich eine Erklärung der ersten Programmzeile, der Anweisung

```
/* Wertetabelle für X-Quadrat */
```

Dieser Befehl entspricht einer **REM-Zeile** in BASIC. Sie könnten in unser BASIC-Programm zur "while"-Schleife die folgende Zeile einfügen:

```
5 REM Wertetabelle für X-Quadrat
```

Der Text zwischen den sog. Slashes (Schrägstrichen) und dem Multiplikationszeichen wird vom Compiler bei der Übersetzung des Programms nicht berücksichtigt.

Vor allem um den Überblick in größeren Programmen zu behalten, sollten Sie sich frühzeitig angewöhnen, Kommentare zur Struktur in Ihre Funktionen (Unterprogramme) einzufügen.

Die Syntax dieser Anweisung lautet immer:

```
/* Kommentartext */
```

Beachten Sie, daß kein Semikolon auf die Kommentarzeile folgen darf.

2.6. DIE DATENEINGABE

Sie sind nun schon in der Lage, die ersten Programme zu schreiben. Doch ein sehr wichtiges Element, das noch fehlt, ist die Möglichkeit einer Dateneingabe.

Ähnlich wie bei der "printf"-Anweisung, die erheblich mehr Möglichkeiten als der normale BASIC-PRINT-Befehl bietet, gibt

es auch bei der Dateneingabe verschiedene Realisierungsmöglichkeiten mit unterschiedlichen Eigenschaften.

Wir werden zunächst, um die Sache für den Anfang einfach zu halten, den BASIC-INPUT-Befehl auf den folgenden Seiten durch zwei unterschiedliche 'C'-Befehle realisieren.

2.6.1 DIE "GETCHAR()"-ZUWEISUNG

Das Schema der ersten Dateneingabe-Möglichkeit in 'C' lautet:

```
buchstabe = getchar()
```

Diese Anweisung, bei der der Variablen "buchstabe" ein Zeichen zugewiesen wird, das über die Tastatur eingelesen werden muß, ermöglicht eine problemlose Eingabe eines einzigen Buchstabens in ein laufendes Programm. Sie ist grob vergleichbar mit einer BASIC-Routine, die einen **GET-Befehl** enthält:

```
10 LET BU$=""
20 GET BU$
30 IF BU$="" THEN GOTO 20
40 :
50 REM BUCHSTABE BU$ ZUGEWIESEN
```

Längere Texte müssen duch eine Wiederholungs-schleife zugewiesen werden. Dies kann in 'C' beispielsweise so aussehen:

```
#include "stdio.h"
main()
{

   int b;
       b = getchar();
       while (b != EOF)
             {
```

```
            printf("%c\n", b);
            b = getchar();
        }
    gemdos(0x1);

}
```

Erklärungen zum Programm:

Das Programm nimmt über die Tastatur eingegebene Buchstaben an und gibt sie doppelt untereinander auf dem Bildschirm aus. Die Eingabe wird mit **CTRL-Z** beendet.

Die Variable "b" beinhaltet eine Zahl vom Typ Integer. Dies mag für einen BASIC-Programmierer befremdend wirken. Eigentlich müßte man einen Character-Variablentyp erwarten, da ein Zeichen eingelesen werden soll.

Über **"getchar()"** wird jedoch genaugenommen nicht das der betätigten Taste entsprechende Zeichen, sondern der ASCII-Wert dieses Zeichens eingelesen. Und dieser Wert muß an eine Variable von Typ **"int"**, nicht aber **"char"** übergeben werden.

Allerdings kann **"b"** auch als Character-Variable definiert werden, das Programm würde trotzdem in fast allen Fällen anstandslos funktionieren. Doch kann es manchmal in der Praxis einige Probleme geben.

Verwenden Sie daher besser Integer- anstatt Character-Variablen zusammen mit der Anweisung "getchar()".

Die Anweisung "%c" in diesem "printf"-Befehl teilt dem Computer mit, daß ein Character, also ein Zeichen ausgegeben wird.

Interessant ist noch das **EOF**. Es bedeutet **"End Of File"** und zeigt an, wann die Eingabe beendet wurde, wann also ein Return erfolgte.

Ein weiterer Hinweis: "!=" steht in 'C' für "**ungleich**", vergleichbar mit dem Äquivalenzoperator "<>" in BASIC.

Der Befehl

```
#include "stdio.h"
```

bewirkt, daß die Standardbibliothek des Compilers in das Programm eingebunden wird. Diese definiert eine Reihe von Standardbefehlen und Konstanten. In unserem Beispiel sind dies die EOF-Systemkonstante und die Funktion "getchar()".

2.6.2. "GETCHAR()" AUF DEM DIGITAL-'C' DES ATARI ST

Sollten Sie das DIGITAL-'C' auf dem ATARI ST verwenden, so ist auf jeden Fall bei der Version, die dem ATARI-Entwicklungssystem beigelegt ist, eine Änderung nötig. Diese Version beinhaltet keine lauffähige "getchar()"-Funktion. Diese muß durch eine gesonderte Funktion nachgebildet werden.

Fügen Sie dazu die folgenden Zeilen an das Ende des obigen Programms an:

```
getchar()
{
     char c;
     return((read(0, &c, 1) > 0) ? c & 0377 : EOF);
}
```

Bei allen folgenden Programmen, die die "getchar()"-Funktion anwenden, wird diese Erweiterung jeweils zusätzlich an das Programmende angefügt.

Zum Zeitpunkt der Veröffentlichung dieses Buches ist nicht sicher, ob die Funktion in der offiziellen DIGITAL-'C'-ST-Version implementiert wird. Ist dies der Fall, so ist die obige Programmerweiterung nicht notwendig.

Auf dem ST ist das Signal zum Zeilenende ein CTRL-Z. Zur Beendigung des Programmablaufs müssen Sie daher nicht RETURN, sondern dieses Control-Zeichen eingeben.

Eine Eingabe bei dem DIGITAL C kann beispielsweise so aussehen:

```
abcd^Z
```

Die Ausgabe lautet dann

```
a
b
c
d
```

Die Funktion dieser weiteren Funktion muß Sie zu diesem Zeitpunkt nicht beschäftigen. Diese und viele weitere Funktionen sind in dem Buch "Tips & Tricks zu 'C' auf dem ATARI ST" in dem Abschnitt zu DIGITAL C ausführlich erläutert.

2.6.3 DIE "SCANF"-ZUWEISUNG

Sie können nun bereits einzelne Zeichen in den Computer eingeben. Dieser Vorgang ist recht einfach zu bewerkstelligen, wie Sie gesehen haben.

Das reicht natürlich nicht aus. Wir wollen uns jetzt damit beschäftigen, wie BASIC-Befehle wie "INPUT A" oder "INPUT A$" in 'C' realisiert werden.

Übersetzen wir das folgende BASIC-Programm

```
10 INPUT A% : REM NUR INTEGER ZAHLEN
20 :
30 END
```

in 'C', so erhalten wir:

```
main()
{

    int a;
    scanf("%d", &a);
    gemdos(0x1);

}
```

Beachten Sie bitte den **Index** der "&a"-Variablen. Dieser wird immer benötigt, wenn eine **numerische** Eingabe über die "**scanf**"-Funktion erfolgt.

Es handelt sich hierbei um einen sog. **Pointer**, der dem Computer mitteilt, wo er die Eingabe zu speichern hat. Damit werden wir uns jedoch später noch genauer beschäftigen.

2.6.4. "SCANF()" AUF DEM DIGITAL-'C' DES ATARI ST

Das DIGITAL-'C' speichert ASCII-Eingaben, also jegliche in der '**scanf()**'-Funktion vorgenommenen Eingaben als Files mit einem Line Feed nach jeder Eingabe. Ein **CTRL-Z (0x1a)** steht dabei für ein End Of File.

Daher muß jede Eingabe auf 'scanf()' mit diesem Control-Code abgeschlossen werden. Dies reicht jedoch zur Annahme der Eingabe nicht aus. Vor dem abschließenden CTRL-Zeichen muß noch ein **formatfremder Character** eingegeben werden.

Dies liegt daran, daß die scanf()-Funktion dann verlassen wird, sobald die Eingabe dem Format nicht mehr entspricht. Die Eingabe der Integerzahl 1245 muß dann beispielsweise folgendermaßen erfolgen:

```
1245_^Z
```

Dabei ist das Leerzeichen '_' das formatfremde Zeichen, welches keine Zahl darstellt, und ^Z stellt das abschließende CTRL-Z Zeichen dar. Nach dieser Eingabe muß RETURN betätigt werden.

Dieser umständliche Eingabevorgang ist ein für den Anwender unzumutbarer Zustand. Es bleibt zu hoffen, daß die offizielle Version des DIGITAL-'C' eine Standardeingabe mittels eines einfachen RETURN zum Abschluß der Eingabe vorsieht.

2.6.5. DIE WEITERE VERWENDUNG DER VARIABLEN IM PROGRAMM

Wenden wir uns der weiteren Verwendung der "&a"-Variablen in einem Programm zu:

Unser BASIC-Programm, das die Variable "A%" nach der Eingabe weiterverwendet, sieht beispielsweise folgendermaßen aus:

```
10 INPUT A%
20 PRINT A%; A%^2
30 END
```

Die in 'C' transformierte Version lautet:

```
main()
{

     int a;

        scanf("%d", &a);
        printf("%d %d\n", a, a * a);

     gemdos(0x1);

}
```

Sie sehen, daß die Verwendung der Variablen "&a" im 'C'-Programm wie in BASIC abläuft.

Der Pointermarker "&" ist also nur im Zusammenhang mit der "scanf"-Anweisung nötig.

2.7 DIE ARITHMETIK IN 'C'

Die Arithmetik wird Ihnen als BASIC-Umsteiger sicher keinerlei Schwierigkeiten machen, weil sie in vielen Punkten fast identisch mit der in BASIC verwendeten ist.

2.7.1 GEMEINSAMKEITEN ZU BASIC

In unseren bisherigen Programmen haben wir die Arithmetik auch wie selbstverständlich, ohne sie zu erklären oder näher darauf eingehen zu müssen, verwendet.

Einige Beispiele, die ich noch einmal kurz aufgreifen möchte, sind:

```
1.  x = x + 1 ;

2.  x * x ;

3.  while(x <= 20)

4.  for(x = 1 ; x <= 20 ; x = x + 1)

5.  a = 2/(3*3)
```

Hier sehen Sie noch einmal, daß Zuweisungen wie a=2/(3*3) in ihrer Syntax sowie in ihrer Logik und der Klammersetzung identisch mit denen in BASIC sind.

Auch die Operatoren

```
<,  >,  <=,  >=,  +,  *,  /,
```

entsprechen denen des BASIC.

2.7.2. UNTERSCHIEDE ZU BASIC

Es gibt jedoch auch kleine Unterschiede, über die besonders BASIC-Programmierer anfangs häufig "stolpern".

Einen solchen Sonderfall haben wir bereits behandelt. Das "ungleich"-Zeichen, das in BASIC durch

```
<>
```

dargestellt ist, wird in 'C' zu

```
!=
```

'C' unterscheidet ferner zwischen

```
=
```

und

```
==
```

Das Zeichen "ist gleich" in dem Ausdruck

```
x = - 1;
```

bewirkt wie in BASIC eine Wertezuweisung der Variablen "x".

In einer "**if**"-Bedingung (siehe auch nächster Abschnitt des Buches) wird mit der Abfrage

```
if (x == 1)
    {
        Befehle
    }
```

mit dem Äquivalenzoperator "**==**" ein Vergleich vollzogen. Es gibt noch eine Reihe weiterer formaler Unterschiede:

Das logische

```
AND
```

des BASIC wird in 'C' durch

```
&&
```

formuliert. Ähnlich verhält es sich mit dem logischen

```
OR
```

welches in 'C' durch

```
||
```

- sogenannte Pipes - dargestellt wird.

Die BASIC-Zeile

```
IF A=1 AND B=2 OR C=5 THEN (...)
```

wird demnach in 'C' zu

```
if(a == 1 && b == 2 || c == 5)
   {
      ...
   }
```

Dies sieht auf den ersten Blick etwas ungewohnt aus. Neu ist jedoch generell nur die Form einiger arithmetischer Ausdrücke in 'C'. Die Syntax sowie die Anwendungsmöglichkeiten sind Ihnen aus BASIC bekannt und erfordern keine großen Umstellungen.

2.7.3 DIE INKREMENT- UND DEKREMENT-OPERATOREN

In 'C' gibt es Anweisungen wie

```
y = x++;
```

Dabei handelt es sich bei dem "++" um einen sog. **Inkrement-Operator**. Er bewirkt, daß der Wert von "x" um eins erhöht wird.

Der Ausdruck

```
y = x++;
```

entspricht demnach dem BASIC-Ausdruck

```
y = x + 1;
```

Genauso verhält es sich mit dem Dekrement-Operator "--":

```
y = x--
```

entspricht exakt der BASIC-Formulierung

```
y = x - 1;
```

Ungewöhnlich ist dabei, daß der Dekrement- oder Inkrement-Operator sowohl vor als auch nach einer Berechnung aktiv werden kann. In

```
y = ++x;
```

wird er beispielsweise vor der Berechnung aktiv. Mehr darüber in dem Kapitel über Arithmetik in diesem Buch.

2.8 WEITERE KONTROLLSTRUKTUREN IN 'C'

Kontrollstrukturen definieren die Ausführungsabfolge von Operationen. Zwei derartige Strukturen, die **"for"** - und **"while"**-Schleife haben wir bereits kennengelernt.

Wenden wir uns in diesem Abschnitt nun kurz der **"if"**- und **"if-else"**-Kontrollstruktur zu und machen uns damit vertraut.

2.8.1 DIE "IF"-ANWEISUNG

Das folgende BASIC-Programm fragt Sie nach einer Zahl und gibt, wenn Sie die Lösungszahl eingeben, eine Meldung aus.

Das Programm sollten Sie nicht zu ernst nehmen, sein Hauptziel besteht darin, die "if"-Struktur zu veranschaulichen.

```
10 INPUT X%
20 :
30 IF X%=13 THEN PRINT "SIEG!":
   PRINT "DAS WAR DIE LÖSUNG"
40 END
```

Transformieren wir dieses Programm in 'C', so erhalten wir:

```
main()
{

    int x;
    scanf("%d", &x);

    if(x == 13)
        {
            printf("Korrekt!\n");
            printf("Das war die Lösung!\n");
        }
    gemdos(0x1);

 }
```

Sie sehen, es gibt mit dieser normalen "if"-Kontrollstruktur für BASIC-Programmierer keinerlei Schwierigkeiten. Der Aufbau ist identisch und die Syntax entspricht der der bereits erörterten "while"-Anweisung.

Achten Sie jedoch besonders auf den Vergleich (x == 13), der in 'C' im Gegensatz zu BASIC mit zwei Gleichheitszeichen formuliert wird.

Auch hier folgt auf den "if"-Befehl kein Semikolon.

Folgen wie in unserem Beispiel **mehrere Befehle** auf die "if"-Bedingung, so müssen diese in geschweifte Klammern eingegliedert werden.

Dies kann jedoch bei nur einer einzigen nachfolgenden Anweisung entfallen.

2.8.2 DIE "IF-ELSE"-ANWEISUNG

Nur wenige BASIC-Versionen besitzen die Möglichkeit von IF-THEN-ELSE-Kontrollstrukturen.

Erweitern wir das obige Programmbeispiel um eine "ELSE"-Anweisung, so erhalten wir:

```
10 INPUT X%
20 :
30 IF X%=13 THEN PRINT "O.K.!":
            PRINT "DAS WAR DIE LÖSUNG!"
              ELSE PRINT "DAS WAR LEIDER
              FALSCH..."
40 END
```

Unser erweitertes 'C'-Programm sieht dann folgendermaßen aus:

```
main()
{
   int x;
     scanf("%d", &x);

      if(x == 13)
```

```
        {
            printf("Ok!\n)
            printf("Das war die Lösung!\n")
        }

    else
        {
            printf("Das war leider falsch...");
            printf("\n");
        }
    gemdos(0x1);

}
```

Wie in BASIC wird die **"else"**- einfach der **"if"**-Anweisung hintergestellt.

Es ist jedoch sehr einfach möglich, einen großen Befehlsblock als "else"-Teil direkt zu formulieren. In BASIC, wo alle "else"-Anweisungen in einer Zeile stehen müssen, ist dies nicht ohne weiteres möglich.

Die einzige Lösungsmöglichkeit besteht in dem Aufrufen einer gesonderten Unterroutine über ein **GOSUB**. Bei einmaligem Aufruf ist dies jedoch nicht so elegant bzw. effektiv.

Die Syntax der "else"-Anweisung in 'C' sollte Ihnen mit Ihrer bisherigen 'C'-Erfahrung selbstverständlich und einleuchtend sein.

Nach dem "else" folgt kein Semikolon, wie auch bei der "while"- oder der "if"- Anweisung.

Alle auf den "else"-Teil folgenden Befehle werden in geschweifte Klammern eingefügt.

Auch hier können diese entfallen, wenn nur ein Befehl nach dem "else" ausgeführt werden soll. Ein Beispiel dafür ist eine printf("Text")-Anweisung.

2.9 DATENTYPEN IN 'C'

2.9.1 VARIABLEN

Die wichtigsten Variablentypen haben Sie bereits kennengelernt:

int
für ganzzahlige **Integerwerte,**

float
für **Gleitkommazahlen** und

char
für **Charakter,** also Zeichenvariablen.

'C' besitzt noch mehr Variablentypen, wie etwa Variablen zum Rechnen mit erhöhter Genauigkeit. Auf alle diese Typen werden wir in den nächsten Kapiteln des Buches noch ausführlich eingehen.

2.9.2 KONSTANTEN

'C' beinhaltet weiterhin die Möglichkeit, symbolische Konstanten zu definieren.

Dies geschieht durch die **"#define"-Konstruktion** und sieht in einem Programm verwirklicht beispielsweise folgendermaßen aus:

```
#define SCHRITTWEITE 1
#define UN.GRENZE 0
#define OB.GRENZE 20
main()
{

    int x;

    for(x = UN.GRENZE; x <= OB.GRENZE;x = x + SCHRITTWEITE )

        printf("%2d\n", x);

    gemdos(0x1);

}
```

Das kurze 'C'-Programm zählt von der unteren Grenze eines Intervalls, festgelegt durch die Konstante **"UN.GRENZE"** mit dem Wert **Null**, bis zur oberen Grenze, **"OB.GRENZE"**, **20**. Die Schrittweite ist auf Eins festgelegt.

Die vor dem Hauptprogramm festgelegten drei Konstanten werden dann in der "for"-Anweisung wieder aufgegriffen.

Sie werden nun sicher nach dem **Nutzen der symbolischen Konstanten** fragen. Gewiß, in diesem kurzen Programm wirken sie recht umständlich und sind eigentlich auch überflüssig.

Der Sinn der Konstanten liegt jedoch vielmehr in deren Anwendung in größeren Programmen. Angenommen, Sie verwenden mehrere "for"-Anweisungen mit den gleichen Parametern, also identischen Intervallgrenzen und Schrittweiten,

so können Sie einfach durch Änderung der Konstantendefinitionen alle Schleifen in Ihrem gesamten Programm neu setzen.

Besonders effektiv ist diese Anwendung, wenn Sie eine Vielzahl an Funktionen, also Unterroutinen einsetzen. Durch die symbolischen Konstanten werden alle Funktionen angesprochen.

Die **Struktur** einer Konstante lautet

```
#define KONSTANTE  Konstanteninhalt
```

Man beachte, daß hinter der "#define"-Konstruktion ebenfalls, wie beispielsweise nach den Kommentaren, kein Semikolon folgt.

Sie können jedoch anstelle von Zahlenwerten als Konstanteninhalt auch Texte verwenden.

Der Compiler setzt die Konstantentexte dann bei der Übersetzung um und ersetzt alle Konstantennamen im Programmtext durch den Text des Konstanteninhaltes.

Voraussetzung dabei ist natürlich, daß der Name der Konstanten nicht innerhalb von Anführungszeichen verwendet wird.

2.9.3 VEKTOREN

Erschrecken Sie nicht bei dem Begriff der Vektoren. In 'C' nennt man so die Ihnen aus BASIC bekannten Datenfelder.

Der BASIC-Dimensionierungs-Befehl

```
10 DIM S(20)
```

wird in 'C' zu der Variablendeklaration

```
int s[20]
```

oder

```
float s[20]
```

bzw. jeder anderen Variablenart, die jedes einzelne Element des Vektors besitzen soll.

Ganz wichtig ist für Sie als durch BASIC "verwöhnter" Programmierer, daß Sie zunächst *jedes* (!) einzelne Vektorelement mit einem Wert belegen müssen.

In BASIC setzt der Befehl

```
DIM S(20)
```

alle Variablen von S(1) bis S(20) automatisch auf Null. In 'C' müssen Sie dies gesondert vollziehen, bevor Sie einen Vektor im Programm weiterverwenden können.

In unserem Beispiel reicht dazu eine kurze "for"-Schleife:

```
main()
{

    int s[20] ;
    int i;
```

```
        for(i = 0; i < 20; i = i + 1)
            s[i] = 0;

            weitere Programmbefehle

        gemdos(0x1);

}
```

In diesem Programm stehen Ihnen dann die Variablen von s[0] bis s[19] innerhalb des Vektors in Ihrem weiteren Programm zur Verfügung.

Zur Übung stelle ich Ihnen nun ein kurzes Programm vor, das die über die **"getchar()"**-Anweisung von der Tastatur eingelesenen Zeichen in einem Vektor ablegt.

In BASIC sähe das so aus:

```
10 DIM T$(50)
20 I=0
30 A$=""
40 GET A$
50 IF A$="" THEN 40
60 :
70 I=I+1
80 T$(I)=A$
90 IF A$<>CHR$(13) AND I<50 THEN GOTO 30
100 END
```

Und hier kommt die 'C'-Version:

```
#include "stdio.h"
main()
{
```

```
    char a;
    int i;
    char t[50];

      for(i = 0; i < 50; i = i + 1)
          t[i] = 0;

      i = 0;

      a = getchar();

        while(a != EOF && i < 50)
            {
                i = i + 1;
                t[i] = a;
                a = getchar();
            }
       printf("Inhalt des Strings:\n");

       for(i = 0; i < 50; ++i)
           printf("%c n", t[i]);

       gemdos(0x1);

}

getchar()
{

    char c;
    return((read(0, &c, 1) >0) ? c & 0377 :EOF );

}
```

Alle Strukturen dieses Programms wurden bereits in dem Abschnitt Dateneingabe-/"getchar()"-Anweisung näher erläutert.

Neu implementiert ist hier der String-Zuweisungsteil. Alle Erklärungen zu der Syntax und den Details zu Vektoren wurden auf den vorherigen Seiten gegeben.

Sie sollten das obige Programm nicht nur lesen, sondern auch in den Computer eingeben. Außerdem sollten Sie es ändern oder erweitern und anhand Ihrer bisherigen 'C'-Erfahrungen ein wenig experimentieren.

'C' lernt man wie jede neue Sprache duch aktives Programmieren und praktische Anwendung, nicht aber durch einfaches Lesen!

Dieser Hinweis gilt übrigens für alle Programme, nicht nur für dieses eine Beispiel.

Auf den folgenden Seiten finden Sie die wesentlichen, in diesem Kapitel behandelten Grundelemente von 'C' noch einmal im Überblick.

Kapitel 3:

Zusammenfassung der Grundelemente von 'C'

Sie haben nun die grundlegenden Strukturen der Programmiersprache 'C' kennengelernt und sind bereits in der Lage, Ihre ersten 'C'-Programme selbständig zu erstellen.

Zunächst sollten Sie Ihr 'C'-Basiswissen festigen. Das bedeutet, daß es am sinnvollsten ist, zu experimentieren, sich jetzt gleich noch einmal vor den Computer zu setzen und einfache 'C'-Programme anhand des in den bisherigen Kapiteln vermittelten Grundwissens zu schreiben.

Die Zeit, die Sie hier zusätzlich investieren, sparen Sie später doppelt ein, wenn wir in den nächsten Abschnitten die komplizierteren Strukturen von 'C' im Detail erarbeiten.

Um Ihnen für Ihre ersten eigenständigen Programmier-schritte in 'C' eine Stütze zu geben, folgt auf den nächsten Seiten eine kurze Übersicht über den bisher vermittelten Stoff.

Hier wird das gesamte bisher an vielen Beispielen erarbeitete Grundwissen knapp zusammengefaßt und im speziellen Vergleich zu dem Ihnen bekannten BASIC vorgestellt.

3.1 DIE PROGRAMMSTRUKTUR

Ein Programm setzt sich aus einzelnen **Funktionen** zusammen. Jede Funktion, also jedes Unterprogramm, besitzt einen Titel, und ihre Anweisungen sind in geschweifte Klammern eingefügt.

Dazu ein Beispiel:

```
quadrat()
{

     Befehle der Funktion "quadrat"

}
```

Die Funktion mit dem Titel **"main()"** nimmt eine Sonderstellung ein, sie wird von allen Funktionen immer als erste ausgeführt und ruft meistens andere Funktionen auf. Alle einzelnen Befehle innerhalb einer Funktion werden durch Semikolons voneinander getrennt, vergleichbar mit den Doppelpunkten zwischen den einzelnen BASIC-Befehlen.

3.2 KOMMENTARE

Kommentare dienen Erläuterungen im Programm und werden vom Compiler nicht berücksichtigt.

Sie haben die **Syntaxform**:

```
/* Kommentartext beliebigen Inhalts */
```

3.3 BILDSCHIRMAUSGABEN

Texte werden folgendermaßen auf dem Display ausgegeben:

```
printf("Text");
```

Bei diesem Beispiel erfolgt kein "Carriage Return", also kein Zeilenvorschub. Um diesen zu erreichen, muß ein Steuerungszeichen wie folgt an den Text gehängt werden:

```
printf("Text\n");
```

Numerische Ausgaben erfordern eine Festlegung des Variablentyps und können eine Formatanweisung enthalten:

```
printf("%d %f7.2\n",a ,b);
```

Die wichtigsten Variablentypanweisungen lauten

d ... für dezimale Ausgabe, d.h. Ausgabe eines ganzzahligen Werts.

f ... Fließzahlenausgabe.

s ... String-Ausgabe, d.h. Zeichenkettenausgabe.

Das Format "7.2" in dem obigen Beispiel der "printf"-Ausgabe bewirkt einen Raum von 7 Stellen insgesamt und 2 Stellen nach dem Komma für die Variable "b".

3.4 VARIABLEN UND KONSTANTEN

Konstanten werden mit der "#define"-Konstruktion vor dem "main()"-Teil eines Programms festgelegt und gelten für alle Funktionen.

Beispiel:
```
#define konstante 33
```

Auf die Definition folgt kein Semikolon.

Die innerhalb einer Funktion verwendeten Variablen müssen vor der Verwendung zuerst definiert werden. Die wichtigsten Variablentypen lauten:

```
int a;      Festlegung für ganzzahlige Werte.

float a;    Festlegung der Variablen"a" als Gleitkommazahl.

char a;     "a" stellt einen Character, ein einzelnes Zeichen
            dar.
```

Alle Variablentypen sind als Vektoren, also Variablenfelder, definierbar. Dies sieht beispielsweise für ein Feld aus 20 ganzzahligen Werten so aus:

```
int a[20];
```

Alle Variablen sowie alle einzelnen Elemente eines Feldes müssen unbedingt mit einem Wert belegt werden, nachdem sie vorher definiert wurden. Haben Sie die Variable "a" als Integer definiert, so muß in der Funktion eine **Wertezuweisung**, wie z.B.

```
a = 1;
```

folgen.

3.5 SCHLEIFEN

Die "**for**"-Schleife wird in 'C' folgendermaßen dargestellt:

```
for(i = 0; i < 20; i = i + 1)
      {

            Befehle innerhalb der Schleife

      }
```

Die drei Parameter geben dabei den Anfangswert des Schleifenintervalls, den Endwert und die Schrittweite an.

Mehrere Schleifenanweisungen müssen in geschweifte Klammern integriert werden, bei nur einem einzigen Befehl ist dies nich nötig. Nach der "for"-Anweisung folgt kein Semikolon.

Die "**while**"-Schleifen haben folgende Syntax:

```
while(x < 15)
      {

          zu wiederholende Befehlsanweisungen
          x = x + 1;

      }
```

Innerhalb der Schleife wird die Schrittweite, in diesem Beispiel "eins", mit

```
x = x + 1;
```

eingegeben.

3.6 DATENEINGABEN

Einzelne Zeichen können über die "**getchar()**"-Anweisung eingelesen werden, vergleichbar dem BASIC-Befehl GET:

```
int a;
a = getchar();
```

Die INPUT-Anweisung aus BASIC läßt sich durch den "**scanf**"-Befehl ersetzen:

```
int a;
scanf("%d", &a);
```

Wie der "printf"-Befehl muß "scanf" den Variablentyp beinhalten, also "**%d**", "**%f**" oder "**%s**" als wichtigste Typen. Der Pointermarker "**&**" vor der einzulesenden Variablen "a" muß immer in einer "scanf"-Anweisung enthalten sein.

3.7 BESONDERHEITEN DER ARITHMETIK IN 'C'

Ungleich, **<>** in BASIC, wird zu **!=** in 'C'.

Das logische **AND** wird durch **&&**, das **OR** durch **||** repräsentiert.

Der Ausdruck **'y = x + 1;'** läßt sich auch mittels des Inkrement-Operators durch **'y = ++x;'** oder **'y = x++;'** darstellen.

Dasselbe gilt für den Dekrement-Operator. **'y = x - 1;'** kann man durch **'y = --x;'** oder **'y = x--;'** ersetzen.

3.8 DIE "IF-ELSE"-KONTROLLSTRUKTUR

Diese Struktur hat folgende Syntax:

```
if(x == 1)
   {

     dann führe die hier folgenden Befehle aus

   }
else
   {

     führe diese Befehle aus.

   }
```

Wichtig ist hier, daß auf "if" und "else" kein Semikolon folgt und der Vergleich durch die **doppelten Gleichheitszeichen** 'if(x == 1)' durchgeführt wird.

Das waren alle elementaren Bestandteile von 'C' noch einmal knapp und übersichtlich auf einen Blick zusammengefaßt.

Diese Liste erhebt natürlich noch überhaupt keinen Anspruch auf Vollständigkeit, da 'C' noch etliche weitere Elemente enthält.

Vielmehr sollen Sie anhand der Befehle, die speziell für Programmierer mit BASIC-Vorerfahrung zusammengestellt wurden, ein Instrument zum Nachschlagen für die eigene Programmerstellung in die Hand bekommen.

Nutzen Sie diese Liste und beginnen Sie mit der Programmierung. Nur durch Praxis lernen Sie 'C'!

TEIL 2:
DIE BASISELEMENTE VON C

Kapitel 4:

Bildschirm Input-/Output-Operationen

In den nun folgenden Kapiteln werden wir uns gezielt mit den Teilaspekten von 'C' befassen und diese ausführlich und anschaulich anhand vieler Beispiele aus BASIC erörtern.

Sie werden dabei **alle Details** sowie die leistungsfähigen 'C'-Strukturen kennenlernen. Dadurch wird ein tiefer Einstieg in 'C' möglich.

Wir beginnen hier, nachdem Sie in dem Einführungskapitel schon einige elementaren Möglichkeiten der Datenein- und ausgabe kennengelernt haben, mit einer ausführlichen Erläuterung der Input-/Output-Operationen auf dem Display.

4.1 AUSGABE VON TEXTEN AUF DEM BILDSCHIRM

Wir haben das Problem der Textausgabe auf dem Bildschirm in dem vorherigen Kapitel bereits angesprochen. Erinnern wir uns kurz und vertiefen dann anschließend das bisherige Wissen anhand vieler neuer, Ihnen bisher vorenthaltenen Details.

Die BASIC-Zeile

```
10 PRINT "HALLO"
```

wurde von uns folgendermaßen in 'C' umgesetzt:

```
main()
 {

      printf("Hallo\n");
      gemdos(0x1);

 }
```

Dabei diente das Steuerungszeichen '\n' zum Generieren eines **Zeilenvorschubs**. Läßt man dieses Zeichen wie in diesem Beispiel aus

```
main()
 {
     printf("ha");
     printf("llo");
 }
```

und setzt den auszugebenden Text "hallo" aus zwei Wörtern zusammen, so entspricht dies dem BASIC-Programm

```
10 PRINT "HA";
20 PRINT "LLO";
```

In beiden Fällen sieht die Bildschirmausgabe folgendermaßen aus:

```
hallo_
```

Dabei stellt der Strich "_" die Cursorposition nach der Ausführung des Befehl dar.

4.2 AUSGABE VON NUMERISCHEN WERTEN

Wir haben bereits in den vorherigen Kapiteln festgestellt, daß die Anzeige von Zahlenwerten in 'C' komfortabler, aber auch aufwendiger gelöst wurde:

Die BASIC-Routine

```
10 X=3.14
20 PRINT X
```

wird in 'C' zu

```
main()
 {

      float x;
         x = 3.14;

         printf("%f\n", x);
      gemdos(0x1);

 }
```

Wollen wir anstelle von **"3.14"** ganzzahlige Werte ausgeben, also beispielsweise ein Programm folgender Art schreiben

```
10 X%=15
20 PRINT X%
```

so lautet das entsprechende 'C'-Programm

```
main()
{

     int x;
        x = 15;
        printf("%d\n", x);
     gemdos(0x1);

}
```

Wir können dasselbe jedoch auch kürzer formulieren:

```
10 PRINT 15
```

Die entsprechende 'C'-Funktion sieht so aus:

```
main()
{

     printf("%d\n", 15);
     gemdos(0x1);

}
```

Die vollständige und präzise "printf"-Anweisung lautet in schematischer Darstellung:

```
printf("Format-Anweisungen", Argument 1, Argument 2, ...);
```

Als Format-Anweisungen bezeichnen wir Kontrollbefehle, die die Ausgabe steuern. In unseren Programmen wurden die Elemente "%d" für die Integerzahl "15" sowie "%f" für den "float"-Wert "3.14" gewählt. Schauen wir uns jedoch auf den folgenden Seiten die Format-Anweisungen genauer an.

4.3 FORMAT-ANWEISUNGEN

Diese Befehle lassen sich untergliedern in die Formatbestimmungen, die beispielsweise festlegen, wie viele Kommastellen eine Zahlenausgabe besitzt, und in die Umwandlungs-Elemente, die den Ausgabentyp wie beispielsweise **"Integer"**, **"Float"** oder **"String"** bestimmen.

Wenden wir uns zuerst den Umwandlungs-Elementen zu.

4.3.1 UMWANDLUNGS-ELEMENTE

Bisher bekannt sind Ihnen die Indices

f ... für Gleitkommawerte wie z.B. 3.14.

d ... für eine dezimale Darstellung, z.B. 15.

Kurz erwähnt wurde auch der Index

s ... den wir zur Zeichenausgabe verwendeten.

'C'-besitzt jedoch wesentlich mehr Umwandlungsindices und auch zu den uns bereits bekannten sind Ihnen in unserem groben Einsteigerüberblick wichtige Informationen vorenthalten worden.

Daher finden Sie auf der nächsten Seite eine vollständige **Indices-Liste** aller Umwandlungsbefehle in 'C':

4.3.1.1 Numerische Ausgaben:

"%d" Wie in den Beispielen deutlich wurde, ist diese Anweisung für eine **dezimale** Darstellung erforderlich. Geben Sie hier dennoch eine Fließkommazahl aus, so erzeugt dies einen Error. Dieser Fehler ist sehr häufig und schleicht sich vor allem bei der Anwendung von Formeln ein, bei denen sich keine Integerzahl ergibt.

"%u" entspricht %d, nur werden die Werte hier **ohne Vorzeichen** dezimal dargestellt.

"%o" stellt das Argument in **Basis 8** ohne Vorzeichen, d.h. ohne die führende 0, dar.

"%x" wie %o, nur erfolgt hier die Anzeige des Argument in **Basis 16.**

"%f" zur Ausgabe von "float"- oder "double"-(Gleitkommawert mit doppelter Genauigkeit) Argumenten. Die Werte werden **dezimal und mit dem Vorzeichen** ausgegeben.

"%e" entspricht %f, gilt also für "float"-und "double"-Argumente, nur erfolgt hier die Anzeige in der wissenschaftlichen **exponentiellen Form:**

```
"(-)m.nnnnnnE(+-)xx"
```

"%g" eine **Entscheidungsform** zwischen f und e. Die Umwandlung erfolgt entweder über %e oder %f, abhängig davon, welches Format kürzer ist.

4.3.1.2 Zeichenausgaben:

"%c" hier wird das auszugebende Argument wie ein **einziges Zeichen** behandelt.

"%s" stellt einen **String**, eine Zeichenkette, dar. Alle mit Zeichen belegten Positionen des Strings werden ausgegeben.

Umwandlung von Zahlenformaten:

Sollte es Ihnen passieren, daß aus Intergerwerten durch eine Operation Gleitkommazahlenwerte werden, so können Sie eine Integer-Zahl (%d) als Float-Zahl (%f) ausgeben und so einen Error vermeiden.

4.3.2 FORMATBESTIMMUNGEN

Diese geben im Programm an, wie viele Stellen vor und nach dem Komma eine Zahl einnimmt, ob sie links oder rechtsbündig ausgegeben wird, etc.

Einige Formate sind bereits vorgewählt:

"%f" legt die Anzahl der **Nachkommastellen** auf **sechs** fest. Die Stellen vor dem Komma sind dabei beliebig.

"%e" In der exponentiellen Darstellung werden ebenfalls **sechs Nachkommastellen** festgelegt.

Wollen Sie diese Formate ändern oder die anderen Formate festlegen, so geschieht das beispielsweise in der Anweisung:

```
"%7.5f"
```

auf sieben Stellen vor und fünf Stellen nach dem Komma einer "float"-Variable.

Zentrierung der Ausgabe:

Folgt in der Formatanweisung direkt nach dem "%" ein **Minus**-Zeichen, so wird der Text oder die Variable nach **links** ausgerichtet.

Die normale Zentrierung, ohne zusätzliches Minus-Zeichen, erfolgt **rechts** im Feld.

Alle diese Hinweise gelten natürlich nur dann, wenn das auszugebende Argument kleiner als die vordefinierte Breite ist.

Ist das auszugebende Argument größer als das dafür definierte Feld, so wird normalerweise das Argument nach links gerückt und die zusätzlichen Ziffern rechts aufgefüllt.

Steht jedoch der Minus-Marker nach dem Prozentzeichen also z.B. "%-7.2f", so läuft die Auffüllung genau umgekehrt ab.

4.3.3 BEISPIELE ZUR NUMERISCHEN DARSTELLUNG

Befassen wir uns nach der Textausgabe mit der Anzeige von Zahlenwerten auf dem Bildschirm. Nehmen wir die folgende BASIC-Programmzeile als Beispiel

```
10 PRINT 2*13
```

Setzen wir sie in 'C' um, so erhalten wir

```
main()
{

    printf("%d\n", 2*13);
    gemdos(0x1);

}
```

Anstatt eine einzige Zahl auszugeben, können wir also genau wie in BASIC eine Berechnung innerhalb der Ausgabe durchführen.

Dies gilt auch für eine **Aneinanderreihung** verschiedener Berechnungen, wie im folgenden BASIC-Programm:

```
10 PRINT 2*13; 2/3; 3.14*2.222222; 4-2.2
```

In 'C' wird daraus die Funktion

```
main()
{

   printf("%d %f %f %f\n", 2 * 13, 2.0 / 3.0;
          3.14 *2.222222, 4.0 - 2.2);
   gemdos(0x1);

}
```

Wenden wir uns nun noch kurz der Anzeige von **Variablenberechnungen** zu.

Das BASIC-Programm

```
10 A=6
20 B=88
30 :
40 PRINT A/B
```

wird in 'C' zu

```
main()
{

      float a,
            b;

            a = 6;
            b = 88;

            printf("%f\n", a / b);
            gemdos(0x1);

}
```

Es gibt demnach hier keine wesentlichen Unterschiede zwischen der BASIC- und der 'C'-Version.

Achten Sie nur auf die Variablendeklaration und darauf, daß Sie bei Berechnungen **nicht ein neues Format erzeugen**. Dies kann der Fall sein, wenn aus zwei Integer Variablen eine Gleitkommazahl wird. In diesem Fall müssen wir, wie bereits oben angegeben wurde, das Format auf **"%f"** ändern, um einen Error oder ein fehlerhaftes Ergebnis zu vermeiden.

4.3.4 DIE TEXTFORMATIERUNG

Sie können nicht nur Zahlen, sondern auch Texte formatieren.

Wie dies funktioniert, läßt sich am besten an konkreten Beispielen zeigen

Der auszugebende Text sei vom Typ "String" mit dem zwölf Zeichen langen Inhalt "Olaf Hartwig".

Format-Befehl:	Resultat:
%10s	Olaf Hartwig
%-10s	Olaf Hartwig
%20s	_____Olaf Hartwig
%-20s	Olaf Hartwig_____
%20.9s	_____Olaf Hart
%-20.9s	Olaf Hart_____
%.5s	Olaf _____
%-.7s	Olaf Ha_____

4.3.5 WEITERE ANWENDUNGEN VON UMWANDLUNGEN UND FORMATEN

Stellen Sie eine normale "float"-Variable auf dem Bildschirm dar, wie beispielsweise in dem folgenden Programm:

```
main()
{

    float a;
      a = 15;

      printf("%f\n", a);
      gemdos(0x1);

}
```

so erhalten Sie diese Ausgabe:

```
15.000000
```

Die sechs Nullen werden automatisch angezeigt, da der Computer ohne zusätzliche Formatanweisung nach dem % sechs Nachkommastellen einstellt.

Dieser Schönheitsfehler wird so behoben:

```
printf("%.0f\n", a);
```

Alle Nachkommastellen werden dadurch **abgeschnitten.**

Wünschen Sie dagegen eine Darstellung mit **zwei Nachkommastellen**, so muß die "printf"-Zeile natürlich folgendermaßen lauten:

```
printf("%.2f\n", a);
```

Ich möchte Sie in diesem Zusammenhang mit dem "%f"-Format noch auf einen sehr häufigen Fehler aufmerksam machen, damit sich dieser nicht in Ihre Programmen einschleicht.

Wir haben die BASIC-Zeile:

```
10 PRINT 2/3
```

Das entsprechende 'C'-Programm erzeugt eine fehlerhafte Ausgabe mit dem Ergebnis **"0.000000"**:

```
main()
{

      printf("%f\n", 2/3);
      gemdos(0x1);

}
```

In BASIC werden die Zahlenwerte "2" und "3" bei einer Division automatisch als Gleitkommazahlen behandelt und nicht als ganzzahlige Integerwerte. Dies müssen wir auch in unserem 'C'-Programm berücksichtigen und es korrekt folgendermaßen formulieren:

```
main()
{

     printf("%f\n", 2.0/3.0);
     gemdos(0x1);

 }
```

Das Ergebnis dieses Programms lautet nun korrekt

```
0.666667
```

4.4 AUSGABE VON TEXT-VARIABLEN AUF DEM BILDSCHIRM

Wie man Zahlenvariablen ausgibt, haben wir bereits weiter vorne ausführlich dargestellt.

Uns geht es nun um die Anzeige von Textvariablen. Ein Beispiel stellt das folgende BASIC-Programm dar:

```
10 A$="ATARI ST"
20 PRINT A$
```

In 'C' sieht das so aus:

```
main()
{

      char *a;
      a = "ATARI ST";

        printf("%s\n", a);
        gemdos(0x1);

 }
```

Die Stringvariable "a" muß hier einen Pointermarker, ein Multiplikationszeichen enthalten, ähnlich dem "&"-Marker bei der "scanf"-Anweisung.

In unserem Fall muß in "printf" ein **Stringvariablentyp ("%s")** verwendet werden.

Das Programm könnte auch kürzer geschrieben werden:

```
main()
{

      printf("%s\n", "ATARI ST");
      gemdos(0x1);

}
```

Nun kombinieren wir die Ausgabe **zweier** verschiedener **Variablentypen.** Hier wieder unser umzusetzendes BASIC-Programm:

```
10 A%=12345
20 B$="DIE ZAHL IST...>"
30 :
40 PRINT B$;A
```

Und nun die Verwirklichung in der 'C'-Version:

```
main()
{

    int a;
    char *b;

       a = 12345;
       b = "Die Zahl ist...>";

       printf("%s %d\n", b, a);
       gemdos(0x1);

}
```

Dieses Programm führt genau das gleiche aus, wie die BASIC-Version. Die Variable **"a"** wird als **Integer**, **"b"** als **String** definiert und beide anschließend mit Werten belegt.

Beachten Sie auch die **Leerstelle** zwischen den beiden Formatanweisungen "%s" und "%d".
Da diese Leerstelle innerhalb der Anführungszeichen steht, wird sie auch auf dem Bildschirm ausgegeben. Das bedeutet, daß zwischen dem String und der Zahl eine Leerstelle steht.

In BASIC erfolgt diese Trennung in der Anweisung

```
40 PRINT A$;B
```

durch die aufeinanderfolgenden Variablen automatisch, in 'C' geschieht dies generell mittels des eingefügten Leerzeichens.

Lassen Sie das Leerzeichen einmal aus. Das Resultat ist, daß beide Werte wie eine einzige Variable, also zusammenhängend, ausgegeben werden.

Sehen wir uns dies in einem weiteren Programm einmal genauer an. Hier die BASIC-Version:

```
10 T$="USER"
20 PRINT "HALLO ";T$
```

Und nun die 'C'-Realisierung:

```
main()
{

    char *t;
    t = "User";

       printf("Hallo %s\n", t);
       gemdos(0x1);

}
```

An diesem Beispiel können Sie gut sehen, daß alle Zeichen innerhalb der Anführungszeichen auf dem Bildschirm ausgegeben werden.

Auffallend ist, daß in der Zeile

```
printf("Hallo %s\n", t);
```

die Reihenfolge der Formatanweisungen und des zusätzlichen Textes beliebig wählbar ist.

Dies läßt sich beispielsweise ausnutzen, wenn Sie zunächst die Variable und dann den Text innerhalb der Anführungszeichen ausgeben wollen.

Dazu ist folgende Zeilenänderung erforderlich:

```
printf("%s Hallo\n", t);
```

Die Ausgabe sieht dann so aus:

```
User Hallo
```

Das Leerzeichen vor dem "Hallo" in den Anführungszeichen wird ebenfalls ausgegeben.

4.4.1 AUSGABE EINES CHARACTERS

'C' unterscheidet im Gegensatz zu BASIC zwischen einzelnen Characters und String-Variablen.

Das BASIC-Programm

```
10 T$="W"
20 PRINT T$
```

sieht in 'C' so aus:

```
main()
{
      char t;
      t = 'W';

      printf("%c\n", t);
      gemdos(0x1);
}
```

Achten Sie auf den Befehl

```
t = 'W';
```

Dies ist die korrekte Syntax für **Character-Variablen**. Wäre "t" ein **String**, stünde dort

```
t = "W";
```

Wo liegt der **Unterschied** zwischen beiden Versionen?

Eine Character-Variable enthält nur ein einziges Zeichen, das in die Hochkommazeichen eingeschlossen ist. Dagegen besitzt ein String noch ein zusätzliches Steuerungszeichen, "\0", das das String-Ende anzeigt.

Der String der Anweisung "t = "W";" besteht in Wirklichkeit aus "W\0" und nicht nur aus 'W'.

4.4.2 WEITERE BILDSCHIRMAUSGABEN

Befassen wir uns zunächst mit der BASIC-Anweisung **CHR$()**. Diese läßt sich in 'C' leicht darstellen.

Zunächst wieder ein BASIC-Programm:

```
10 PRINT CHR$(67)
```

Und hier die 'C'-Version:

```
main()
{

     printf("%c\n", 67);
     gemdos(0x1);

}
```

Diese Routine gibt den Buchstaben mit dem ASCII-Wert 67 auf dem Bildschirm aus. Dies entspricht dem Wert des Buchstaben "C".

Die Anweisung CHR$() wird durch den Umwandlungsteil des "printf"-Befehls ersetzt.

Auch die Funktion ASC("") läßt sich ähnlich in 'C' formulieren:

Das BASIC-Programm

```
10 PRINT ASC("B")
```

gibt den ASCII-Wert des Buchstaben "B" auf dem Bildschirm aus. Dieser lautet 66.

Die entsprechende 'C'-Funktion lautet:

```
main()
{

     printf("%d\n", 'B');
     gemdos(0x1);

}
```

Eine andere, etwas ausführlichere Version ist die folgende:

```
main()
{

    char t;
    t = 'B';
       printf("%d\n", t);
       gemdos(0x1);

}
```

Diese beiden Funktionen, also die Entsprechungen der ASC- und der CHR$- Funktionen, werden Sie in 'C' sehr sinnvoll zur Analyse von Strings oder Eingaben einsetzen können, um festzustellen, welche Zeichen enthalten sind, oder um spezielle Zeichen zu eliminieren.

4.4.3 ZUSÄTZLICHE AUSGABEMÖGLICHKEITEN

Bisher haben wir lediglich die "printf"-Funktion zur Anzeige von Daten verwendet. Es gibt jedoch noch zwei weitere Funktionen zur Datenausgabe.

Dies sind die Anweisungen

```
putchar()
```

und

```
puts()
```

Was leisten diese Funktionen?

"putchar()" ist direkt mit der im Einführungskapitel behandelten Funktion

```
a = getchar();
```

verbunden.

Sie erinnern sich sicher noch, dieser Befehl bewirkt das Einlesen eines einzelnen Zeichens, eines Characters.

Entsprechend gibt "putchar()" ein einzelnes Zeichen auf dem Bildschirm aus.

Die Verwendung wird am besten an einem konkreten Beispielprogramm deutlich:

```
#include "stdio.h"
#define putchar(a) putc(a,stdout)
main ()
{

        char a;
           a = 'P';

           putchar(a);
        gemdos(0x1);

}
```

Diese Routine liest 'P' in die Variable "a" ein und gibt dieses einzelne Zeichen über die Anweisung "putchar()" auf den Bildschirm aus.

Mittels **"putchar()"** können Sie demnach ohne den Umwandlungsaufwand der **"printf"**-Funktion effektiv einzelne Zeichen ausgeben. Damit beschränkt sich jedoch schon die Anwendung dieses Befehls zur Bildschirmausgabe.

Nun zur Anweisung "**puts()**". Dieser Befehl steht für "put string", d.h. hiermit können Zeichenketten ausgeben werden.

Dazu ein Beispiel: Das BASIC-Programm

```
10 A$="OLAF HARTWIG"
20 PRINT A$
```

läßt sich mittels der "puts()"-Funktion in 'C' fast genauso einfach formulieren:

```
#include "stdio.h"
main()
{

      char *a;
      a = "olaf hartwig";

      puts(a);
      gemdos(0x1);

}
```

Hier nun zum Vergleich unsere alte Formulierung mittels "printf":

```
main()
{

     char *a;
     a = "olaf hartwig";

     printf("%s\n", a);
     gemdos(0x1);

}
```

Sie können die beiden Beispiele folgendermaßen auch kürzer anwenden. Angenommen, Sie wollen die BASIC-Zeile

```
10 PRINT "P"
```

nach 'C' transformieren, so ist diese Formulierung möglich:

```
#include "stdio.h"
#define putchar(c) putc(c,stdout)
main()
{

      putchar('P');
      gemdos(0x1);

}
```

Zum Vergleich das gleiche Programm, realisiert mit der herkömmlichen **"printf"**-Anweisung:

```
main()
{

      printf("%c\n", 'P');
      gemdos(0x1);

}
```

Ähnliches gilt für den Ausdruck von Strings. Die BASIC-Zeile

```
10 PRINT "OLAF HARTWIG"
```

wird in 'C' durch die Anwendung der Anweisung **"puts()"** zu folgender Mini-Funktion:

```
#include "stdio.h"
main()
{

     puts("olaf hartwig");
     gemdos(0x1);

}
```

Vergleichen Sie auch diese kurze Routine einmal mit dem entsprechenden 'C'-Programm mit **"printf"**

```
main()
{

     printf("%s\n", "olaf hartwig");
     gemdos(0x1);

}
```

Sie sehen, daß die Verwendung der Funktion **"puts()"** und **"putchar()"** zur Ausgabe von Zeichen verglichen mit der normalen, aufwendigeren **"printf"**- Anweisung viel eleganter ist.

In der Programmierpraxis werden diese beiden kompakten Befehle ihre Anwendung finden, wenn keinerlei Umwandlungsformate erforderlich sind, der auszugebende Text also nicht formatiert auf dem Bildschirm ausgegeben werden soll. *Für die Formatierung mittels der Formatanweisung des "printf"-Befehls gibt es keinerlei Ersatz.*

Auch wenn mehrere Variablen oder Texte in einer Zeile hintereinander ausgegeben werden sollen, ist dies nur über "printf" zu erreichen, da "puts()" oder "putchar()" einen automatischen Zeilenvorschub ausführen.

4.5 DATENEINGABEFUNKTIONEN

Auch hier gibt es zu unseren bereits kurz in dem Einsteigerkapitel erörterten Eingabefunktionen "scanf" und "getchar()" wie bei der Datenausgabe weitere Möglichkeiten zur Eingabe von Daten.

Vor allem sind jedoch noch zusätzliche detaillierte Informationen zu unseren beiden bekannten Eingabeanweisungen erforderlich.

Wenden wir uns zunächst der Funktion "getchar()" zu:

4.5.1 DIE "GETCHAR()"-FUNKTION

Wir haben diese Funktion mit der GET-Anweisung des BASIC verglichen. In einigen BASIC-Dialekten ist dies anstelle von GET auch die INKEY$-Funktion.

Das BASIC Programm:

```
Version mit INKEY$:

10 A$=INKEY$
20 IF A$="" THEN GOTO 10
30 :
40 PRINT A$
```

```
Version mit GET:

10 GET A$
20 IF A$="" THEN 10
30 :
40 PRINT A$
```

wird in 'C' zu dem folgenden Programm:

```
#include "stdio.h"
#define getchar() getc(stdin)
#define putchar(c) putc(c,stdout)
main()
{

    int a;
    a = getchar();

    while(a != EOF)
      {
          putchar(a);
          a = getchar();
      }
    gemdos(0x1);

}
```

Erklärungen zu den speziellen Strukturen wurden bereits im Einführungskapitel gesehen. Um unnötige Wiederholungen zu vermeiden, verzichten wir hier auf eine nochmalige Erklärung.

Vergleichen Sie jedoch dieses Programm einmal mit unserem "getchar()"-Programmbeispiel in der Einführung.

In dieser hier gewählten Formulierung wurde bewußt die "putchar()"-Anweisung zur Ausgabe des über "getchar()" eingelesenen Zeichens gewählt. Ihnen sollte deutlich werden, daß beide Befehlen miteinander verwandt sind.

Die **"putchar()"-Syntax** lautete:

```
putchar(Buchstabe);
```

Entsprechend sieht die **"getchar()"**-Struktur aus:

```
Buchstabe = getchar();
```

4.5.2 DIE "GETS()"-EINGABE

Im Zusammenhang mit der Anweisung "putchar()" zur Ausgabe eines einzelnen Zeichens haben wir die Funktion

```
puts()
```

zur **Anzeige eines Strings** erörtert.

Den entsprechenden Befehl gibt es auch zur **Dateneingabe**. Er lautet:

```
gets(a);
```

"gets" ist die Abgekürzung für "get String". Er liest eine Zeichenkette, also einen Text, in die in Klammern gesetzte Variable ein.

Schauen wir uns dazu einmal ein Beispiel zur Verdeutlichung an:

```
10 INPUT A$
20 PRINT A$
```

Dies wird in 'C' mittels der Anweisungen "gets()" und "puts()" so formuliert:

```
#include "stdio.h"
main()
{
      char *a;

      gets(a);
      puts(a);
      gemdos(0x1);
}
```

Beachten Sie, daß der Befehl "puts()" zur Ausgabe eines Strings eleganter, weil einfacher und **unkomplizierter**, als die "printf"-Version ist, die hier zum Vergleich folgt:

```
#include "stdio.h"
main()
{

      char *a;

      gets(a);
      printf("%s\n", a);
      gemdos(0x1);

}
```

Halten wir als Fazit fest:

Zum Einlesen von einzelnen Zeichen oder von Strings, Zeichenketten, also sind die Anweisungen "getchar()" und "gets()" beim Programmieren dem "scanf"- Befehl vorzuziehen.

Hier gilt also entsprechendes, wie schon bei der Erörterung von **"putchar()"** und **"puts()"** im Bezug auf effektives Programmieren gesagt wurde.

Bei der DIGITAL-'C'-Version des ST-Entwicklungssystems treten mit der 'gets()'-Funktion **Probleme** auf. Diese sind durch das fehlerhafte und in keiner Weise dem üblichen 'C' Standard entsprechenden Eingabeformat bedingt. Bei allen anderen ST-'C'-Compilern funktioniert diese Funktion problemlos. Es bleibt zu hoffen, daß die kommerzielle Version des DIGITAL 'C' über ein Standard-Eingabeformat verfügt.

4.5.3 DIE "SCANF"-EINGABEANWEISUNG

Dieser Befehl läßt sich nicht nur zur Eingabe numerischer Werte, also Zahlen, verwenden, sondern ermöglicht auch die Eingabe von Texten oder einzelnen Zeichen.

Aufgrund dieser Universalität gleicht diese Anweisung dem entsprechenden BASIC-Befehl "INPUT" mehr als alle anderen 'C'-Funktionen.

Diese vielfältigen Einsatzmöglichkeiten werden jedoch durch eine etwas kompliziertere Struktur "erkauft".

Wir werden die beiden möglichen Anwendungsarten im folgenden getrennt und detailliert erörtern. Zunächst wenden wir uns der Eingabe von Zeichen und Strings zu, danach befassen wir uns mit numerischen Daten. Um ferner auch Zahlen in ein 'C'-Programm einzugeben, bietet die **"scanf"**- Anweisung die beste Möglichkeit zur Realisierung.

4.5.3.1 "SCANF" ZUR ZEICHEN- UND STRING-EINGABE

Das BASIC-Programm

```
10 INPUT T$
20 PRINT T$
```

das einen String einliest und ausgibt, läßt sich abgesehen von der "puts()"-Anweisung, auch mittels des Befehls "scanf()" realisieren, wie in dem folgenden 'C'-Programm gezeigt wird:

```
main()
{

     char *t;
     scanf("%s", t);

     printf("%s\n", t);
     gemdos(0x1);

}
```

Zunächst wurde die Variable "t" durch den Pointermarker "*" als String deklariert. Dadurch ist es nicht mehr erforderlich, den Pointer in die "scanf"-Anweisung aufzunehmen. Es handelt sich ja bereits um eine Zeigervariable.

Vielleicht erinnern Sie sich, bei Zahleneingaben mußten wir den Pointermarker "&" in den "scanf"- Befehl implementieren.

Generell kann man festhalten, daß alle Variablen innerhalb eines "scanf"-Befehls Pointer sein müssen.

Wichtig ist in unserem Demo-Programm vor allem die Kontrollanweisung "%s" innerhalb von "scanf". Diese teilt wie bei "printf" dem Befehl mit, daß nun eine Stringanweisung folgt.

Das DIGITAL-'C'-spezifische Eingabeformat wurde bereits in Einsteigerkapitel erörtert. Die Eingabe der Zahl 15 erfolgt beispielsweise als

```
15_^Z
```

und die Eingabe eines Strings wird mit **CTRL-Z** beendet:

```
eingabetext^Z
```

4.5.3.2 ARRAYS ANSTELLE VON POINTER

Wir hätten das Programm auch ohne die Pointervariable formulieren können, wie in diesem Beispiel dargestellt:

```
main()
{

    char t[30];
    scanf("%s", t);

    printf("%s\n", t);
    gemdos(0x1);

}
```

Hier wird die Character-Variable "t" als Feld mit 30 Elementen deklariert.

Ein Array (Feld) wird wie eine Pointervariable behandelt. Das bedeutet, daß Sie nun auch Felder in Verbindung mit "scanf"-Befehlen verwenden können.

Allerdings ist die Pointerkonstruktion **flexibler** und vor allem **speicherplatzsparender** als die Array-Struktur.

Dies liegt daran, daß auch bei einer Eingabe von beispielsweise zwölf Zeichen der Array die Länge der 30 festdefinierten Elemente besitzt.

Dagegen ist eine Pointervariable nur zwölf Zeichen plus ein Endmarker lang. Sie sparen vor allem bei vielen Arrays oder Pointern sehr viel Speicherplatz.

Wenden wir uns nun einem weiteren kleinen Problem zu:

Das BASIC-Programm

```
10 INPUT "GEBEN SIE DEN TEXT EIN ";X$
20 PRINT X$
```

soll in 'C' umgesetzt werden. Am sinnvollsten geschieht dies folgendermaßen:

```
#include "stdio.h"
main()
{

    char *x;

       printf("Geben Sie den Text ein\n");

       scanf("%s", x);
       puts(x);
       gemdos(0x1);

}
```

Die beiden Programme leisten exakt dasselbe. Die Eingabe erfolgt in beiden Fällen direkt nach dem Text, ohne daß ein Zeilenvorschub ausgeführt wird.

4.5.3.3 DIE EINGABE VON ZAHLEN ÜBER "SCANF"

Das BASIC-Programm

```
10 INPUT Z%
20 PRINT Z%
```

wird folgendermaßen in 'C' umgesetzt:

```
main()
{

    int z;

       scanf("%d", &z);
       printf("%d\n", z);
       gemdos(0x1);

}
```

Soweit nichts Neues. Wie in dem Einführungskapitel bereits erläutert, ist der Pointer "&" innerhalb der "scanf"- Anweisung unbedingt notwendig, damit der Computer weiß, wo er den einzulesenden Wert im Speicher ablegen muß.

Erstellen wir nun ein Programm, das sowohl ganzzahlige Werte als auch Gleitkommazahlen annimmt. In BASIC ist dies ganz einfach so möglich:

```
10 INPUT Z
20 PRINT Z
```

Und hier die Realisierung in 'C':

```
main()
{

      float z;

          scanf("%f", &z);
          printf("%f\n", z);
          gemdos(0x1);

 }
```

Dieses Programm wandelt automatisch über die "float"-Variable "z" eingegebene Integerwerte in Gleitkommazahlen um.

Wie in dem Abschnitt über "Formatierung" dieses Buches bereits erwähnt werden die Zahlen dann in dem Format.

```
15.000000
```

im Beispiel der eingegebenen Integerzahl 15 ausgegeben. Wie dies zu korrigieren ist, wurde in dem oben genannten Abschnitt ausführlich erläutert.

Eine Möglichkeit ist die "printf"-Anweisung

```
printf("%.0f\n", z);
```

die die Nachkommastellen der Zahl abtrennt.

4.5.3.4 SIMULTANE EINGABE MEHRERER DATEN

In BASIC haben wir die Möglichkeit, mehrere Variablen in einem INPUT-Befehl gleichzeitig einzulesen:

```
10 INPUT T$, A, B%
20 PRINT T$;A;B%
```

Die Eingabe der einzelnen Variablen erfolgt dabei in BASIC wie in 'C' durch Kommata voneinander getrennt, also beispielsweise

```
text, 2, 3.14
```

Das BASIC-Programm sieht in 'C' mit einer "scanf"-Anweisung folgendermaßen aus:

```
main()
{

      char *t;
      float a;
      int b;

          scanf("%s %f %d", t, &a, &b );
          printf("%s %f %d", t, a, b);

          gemdos(0x1);

}
```

Beide Programme leisten dasselbe. In 'C' müssen nur, wie bereits bekannt, die Formatanweisungen in die "scanf"- und "printf"-Befehle integriert werden.

Wie bei "printf" werden auch bei der "scanf"-Anweisung die einzelnen Format-Anweisungen ("%s %f %d") durch ein Leerzeichen voneinander getrennt dem Variablen-Einlesungsteil (t, a, b) vorangestellt.

4.5.4 DIE GET-/INKEY$-FUNKTION IN 'C'

Sicher werden Sie nun sagen, daß wir diese Funktion doch bereits mit der Anweisung **"getchar()"** behandelt hätten.

Dies ist zwar richtig, aber die **"getchar()"**-Anweisung entspricht nicht genau der BASIC-Zeile

```
10 A$=INKEY$
```

Die Anweisung

```
Buchstabe = getchar();
```

nimmt zwar wie die INKEY$- oder GET-Funktion einen einzigen Buchstaben an, erwartet dann jedoch ein RETURN zur Beendigung der Eingabe.

Dies ist bei der neuen 'C'-Funktion

```
Buchstabe = getch();
```

nicht erforderlich. Der Befehl **"getch()"** entspricht genau der BASIC-Routine:

```
10 A$=INKEY$
20 IF A$="" THEN 10
```

Das bedeutet konkret, daß das Programm, wenn es an diese Anweisung gelangt, solange unterbricht, bis eine Taste betätigt wird. Das dieser Taste entsprechende Zeichen wird dann an **"getch()"** übergeben.

In einem Programm verwirklicht sieht das so aus:

```
#include "stdio.h"
#define putchar(a) putc(a, stdout)
#define getchar() getc(stdin)
main()
{

     int a;

        a = getch();
        putchar(a);
        gemdos(0x1);

}
```

```
char bf[100];
int b=0;
getch()
{

      return((b > 0) ? bf[--b] : getchar());

}
```

Hier wird deutlich, wie einfach die Operation mit der "getch()"-Anweisung ist. Das **DIGITAL 'C'** verfügt über die Funktionen **'putchar()'**, **'getchar()'** und **'getch()'** in der vorliegenden Version noch nicht. Diese Befehle müssen wie in diesem Programm dargestellt nachgebildet werden.

Achten Sie nur darauf, daß die einzulesende Variable von Typ Integer ist. Wie bei der Anweisung "getchar()" liest diese Funktion den ASCII-Wert des eingegebenen Zeichens ein, der natürlich eine Zahl darstellt.

Den Befehl

```
putchar(a);
```

könnte man auch durch

```
printf("%c\n", a);
```

ersetzen.

4.5.5 REALISIERUNG VON PUTCHAR(), GETCHAR() UND GETCH() AUF DEM DIGITAL 'C' DES ATARI ST

Die 'C'-Entwicklungspaketversion beinhaltet diese Funktionen nicht. Sie lassen sich jedoch schnell nachbilden. Die **'getchar()'**-Funktion wurde bereits in dem Einsteigerkapitel auf den ST übertragen.

Wollen Sie diese drei Funktionen auf dem DIGITAL 'C' einsetzen, so ist die folgende Erweiterung an das Programmende anzufügen:

```
/* getchar() */
getchar()
{

      char c;
      return((read(0, &c, 1) > 0) ? c & 0377 : EOF);

}

/* getch() */
char bf[100];
int b = 0;

getch()
{

     return((b > 0) ? bf[--b] : getchar());

 }
```

Die Eingabe bei allen Funktionen muß bei dem DIGITAL 'C' jeweils mit einem **CTRL-Z** beendet werden. Alle anderen bekannten 'C'-Compiler verfahren nach dem üblichen 'C'-Standard und benötigen dieses zusätzliche Steuerzeichen nicht.

Die Funktion 'putchar()' kann als Makro nachgebildet werden. Fügen Sie dazu noch vor die Funktion 'main()' die folgende #define-Anweisung in Ihr Programm ein:

```
#define putchar(c)  putc(c, stdout)
```

Entsprechend zu diesem Makro kann auch die obige 'getchar()'-Funktion als Makro definiert werden. Dieser lautet

```
#define getchar()  getc(stdin)
```

Die Funktion **'putch()'** läßt sich fast immer durch die **'putchar()'**-Funktion ersetzen und ist daher hier nicht nachgebildet, obwohl sie im Befehlssatz des DIGITAL 'C' nicht enthalten ist.

Im Rahmen dieses Buches ist es für Sie an dieser Stelle nicht notwendig, diese Funktionen zu vollständig verstehen. In allen Programmen dieses Buches sind die Befehlserweiterungen einfach in dem jeweiligen Programm auf dem ST eingefügt, soweit diese Funktionen angewendet werden sollen.

Kapitel 5:

Die Variablentypen in 'C'

Wenden wir uns nach den Bildschirm-Input-/Output-Operationen nun den **Datentypen** in 'C' sowie deren weiterem Umfeld zu.

Dazu gehören Erläuterungen zu den erlaubten Variablennamen, den bereits schon am Rande erwähnten Konstanten, den möglichen Datentypen, Datentyp- Umwandlungen und Deklarationsvorschriften.

Anschließend werden wir noch weitere wichtige Bereiche und zwar **Arrays**, also Datenfelder, den unterschiedlichen Einsatz von **globalen und lokalen Variablen** in Programmen sowie **Pointer (Zeiger)** in allen wesentlichen Details und Sonderfällen vorstellen.

Beginnen wir mit den

5.1 VARIABLENNAMEN

Bei BASIC besteht eine Einschränkung von Variablennamen auf nur zwei Buchstaben. Obwohl die Namen durchaus länger sein können, werden nur diese ersten beiden Buchstaben des Namens vom Rechner unterschieden.

Es gibt hier einige wenige Ausnahmen, wie z.B. das BASIC der Sinclair-Computer ZX-Spectrum und QL, die beliebig lange Variablennamen zulassen und auch unterscheiden.

Doch normal ist, daß beispielsweise die Namen

```
VAR_1
```

und

```
VAR_2
```

in BASIC als ein und dieselbe Variable angesehen werden, da die ersten beiden Buchstaben "VA" identisch sind.

'C' geht hier weiter und unterscheidet die ersten acht Buchstaben und Zeichen. Die Namen können jedoch genau wie in BASIC länger sein als diese ersten acht signifikanten Stellen.

Wichtig ist hier genau wie in BASIC, daß das erste Zeichen ein Buchstabe und nicht etwa ein Sonderzeichen oder eine Ziffer darstellen darf.

Das Sonderzeichen "_" zählt dabei als Buchstabe. Es sollte vornehmlich zum Verbinden mehrerer Namen innerhalb einer Variable eingesetzt werden, um eine bessere Lesbarkeit des Namens zu gewährleisten.

Anstatt also beispielsweise die Variable

```
plzort
```

für die Postleitzahl und den Ort in einer Adressdatei zu verwenden, sollten Sie besser den Namen

```
plz_ort
```

wählen.

Ein weiterer wesentlicher Unterschied zwischen BASIC- und 'C'-Variablennamen besteht darin, daß **Groß- und Kleinbuchstaben** unterschieden werden.

Die Variable

```
NAME
```

ist also nicht identisch mit

```
name
```

Vor allem für viele BASIC-Programmierer ist dies zunächst etwas ungewohnt, da BASIC diese Unterscheidung nicht vornimmt.

Die 'C'-Namen haben jedoch deutliche Vorteile:

In 'C' hat es sich eingebürgert, **symbolische Konstanten** immer in **Großbuchstaben** zu schreiben, **alle anderen Variablentypen** dagegen in **Kleinbuchstaben.**

In einem Programm führt diese Regelung zu wesentlich mehr Übersicht, da Konstanten auf diese Weise sofort von Variablen unterschieden werden können. Sie sollten sich dies daher schon jetzt zu eigen machen.

Reservierte Sprachelemente von 'C' dürfen auf keinen Fall als Variablen verwendet werden. Zu diesen Elementen gehören:

```
if              else
while           do while
for             switch
case            default
break           continue
return          goto
```

Beachten Sie auch, daß die **Sprachelemente** generell immer in **Kleinschrift** formuliert werden müssen.

Es ist vernünftig, die Variablennamen derart zu wählen, daß sie einen Sinn ergeben, d.h. es sollte sofort ersichtlich sein, welchen

Zweck die Variable in einem Programm besitzt. Sie ersparen sich so beim "Debugging" umfangreicherer Programme viel Suchzeit.

Vor allem bei BASIC gewöhnt man sich sehr schnell daran, nur aus ein oder zwei Buchstaben bestehende Kurzvariablen zu verwenden. Versuchen Sie sich dieses bei 'C' vor allem in komplexeren Programmen abzugewöhnen.

5.2 KONSTANTEN

Wir haben in unserem Einsteigerkapitel bereits ein Programm entwickelt, das symbolische Konstanten verwendet.

Erinnern wir uns. Die Definition vollzog sich über die Konstruktion

```
define#
```

Die Struktur lautet dabei:

```
define# KONSTANTE Konstanteninhalt
```

Nach dieser Deklaration folgt kein Semikolon!

Bisher haben wir lediglich Integer-Konstanten deklariert. Es lassen sich jedoch alle Variablentypen als Konstanten definieren.

Im folgenden Beispielprogramm verwenden wir zur Veranschaulichung die vier wichtigsten Datentypen in 'C' als Konstanten.

Es handelt sich um **Integer, Float, Char** sowie **Stringvariablen:**

```
define# INTEGER 22
define# FLOAT   1.2345
define# CHAR    'D'
define# STRING "Dies ist ein Text"
main()
{

      printf("%d n", INTEGER);
      printf("%f n", FLOAT);
      printf("%c n", CHAR);
      printf("%s n", STRING);

      gemdos(0x1);

}
```

Wie Sie sehen ist die Anwendung aller Konstanten in einem Programm trotz unterschiedlichen Typs identisch zu der der entsprechenden Variablen.

Wie üblich müssen Texte jedoch in Hochkommata gesetzt werden. Anders als bei der BASIC-Anweisung

```
10 CHAR  ="D"
```

muß die 'C'-Festlegung unbedingt

```
define# CHAR  'D'
```

lauten.

Das Zuweisungsgleichheitszeichen ist also in 'C' nicht erforderlich, außerdem stehen anstelle von Anführungszeichen ("") Hochkommata ('').

Diese Unterschiede werden unter Umständen durch Ihre BASIC-Gewöhnung bei eigenen 'C'-Programmen zu Fehlern führen.

Ein Beispiel einer **fehlerhaften Anweisung** wäre:

```
define# CHAR   "D"
```

Solch ein Fehler, bei dem Hochkommata und Anführungszeichen vertauscht sind, ist besonders ärgerlich, weil er sehr schwierig zu finden ist. Auch wird meistens keine Fehlermeldung generiert, sondern statt dessen die Ausgabe einfach unterdrückt.

Vom BASIC-Standpunkt aus betrachtet, erscheint die Anweisung auch nicht fehlerhaft, da BASIC immer Anführungszeichen verwendet. Achten Sie daher auf diese kleinen Detailunterschiede zwischen BASIC und 'C', Sie können sich auf diese Weise später viel Ärger und Frustationen bei der Fehlersuche ersparen.

String-Konstanten, auch konstante Zeichenketten genannt, werden dagegen in Anführungszeichen eingeschlossen. Dies ist vergleichbar mit String-Variablen:

```
define# STRING "Dies ist ein Text"
```

Dabei gelten die Anführungszeichen nur als Begrenzer für den Text, sie sind jedoch nicht Inhalt der Konstanten.

Soviel zu den String-Konstanten. Bei **"float"**-Konstanten läßt sich der Ausdruck auch wissenschaftlich darstellen.

Beispiele dafür sind

```
define# FLOAT 12.65432E-4
```

sowie

```
define# FLOAT 0.02e5
```

Alle Float-Konstanten werden vom Compiler wie Double-Variablen behandelt, d.h. genau wie Float-Variablen, jedoch mit doppelter Genauigkeit.

Long-Konstanten werden mit einem "L"-Index versehen:

```
12345678L
```

ist ein Beispiel für eine Long-Konstante.

Ist eine Integer-Konstante zu groß für den Typ Integer, so wird sie automatisch als "Long" interpretiert.

Alle die festgelegten symbolischen Konstanten werden bei der Übersetzung durch den Compiler im Programm durch den Inhalt der Konstanten ersetzt und so als feste Werte verwendet.

5.3 DATENTYPEN

Die elementaren Datentypen von BASIC lauten:

```
A%    .....  Integer-Wert
A     .....  Float-Variable
A$    .....  Character-/String-Variable
```

Auf die in der Liste fehlenden Arrays gehen wir später in einem gesonderten Abschnitt ein.

Diese Datentypen finden sich auch in 'C' wieder. Zusätzlich ählt man jedoch auch **"double"**-Werte zu den elementaren Typen.

Die folgende Liste zeigt die in 'C' möglichen Datentypen:

Datentyp:	Erklärung:
int	Ganzzahliger Wertinnerhalbdes von dem Computer als "natürlich" angesehenen Zahlenbereichs.
float	Gleitkommawert mit normaler Genauigkeit.
double	Gleitkommawertmit doppelter Genauigkeit.
char	Ein Byte, jedes Zeichen aus dem Zeichensatz des Computers.

Integer-Variablen lassen sich noch weiter unterteilen. Dazu ein anschauliches kurzes Diagramm:

```
                 --------- 1. short int
                -
        int     ---------- 2. long int
                -
                 --------- 3. unsigned int
```

"Short int" und **"Long int"** stellen Integerzahlen unterschiedlicher Größe dar.

"Unsigned Integer"-Werte befolgen die arithmetischen "modulo 2^n"-Regeln, wobei "n" die Anzahl der Bits in einem Integer-Wert darstellt. Die "Unsigned Int"-Werte sind immer positiv.

Vereinbarungen der erweiterten Integertypen sehen folgendermaßen aus:

```
short int a;
long int b;
unsigned int c;
```

Normalerweise kann das Wort "int" in den Deklarationen auch ausgelassen werden.

An dieser Stelle möchte ich noch einmal den Hinweis wiederholen, daß alle Variablen nach der Deklaration einen Wert zugewiesen bekommen müssen, bevor sie im Programm verwendet werden können.

Die Ihnen aus BASIC bekannten Datentypen STRING und ARRAY werden wir in den nächsten Abschnitten behandeln.

5.4 UMWANDLUNGEN VON DATENTYPEN

Variablen werden und können in BASIC fast nicht in andere Variablentypen umgewandelt werden. In 'C' ist dies anders. Datentyp-Umwandlungen kommen äußerst häufig vor.

Einige Umwandlungen haben wir bereits erwähnt:

5.4.1 CHARACTER-/INTEGER-UMWANDLUNGEN

Im vorherigen Kapitel haben Sie beispielsweise das äquivalente 'C'-Programm zu der BASIC-Anweisung

```
10 PRINT CHR$(66)
```

kennengelernt.

Es lautete folgendermaßen:

```
main()
{

        printf("%c\n", 66)
        gemdos(0x1);

}
```

Das Resultat beider Programme besteht darin, daß das Zeichen "B", das den ASCII-Wert 66 besitzt, auf dem Bildschirm ausgegeben wird.

Hierbei wird in den Programmen, wenn man es genau nimmt, die Zahl 66 in den Buchstaben "B" umgewandelt, sowohl in BASIC als auch in 'C'.

'C' geht jedoch noch weiter. Möglich ist beispielsweise auch das für Sie als BASIC-Programmierer etwas ungewöhnliche Programm:

```
main()
{

        int wert;
        wert = 123 + 'B' ;

        printf("%d\n", wert);
        gemdos(0x1);

}
```

Die Routine wandelt das Zeichen 'B' in der Zuweisung

```
wert = 123 + 'B' ;
```

automatisch in den Integer-Wert seiner ASCII-Zahl um und addiert diesen zu 123.

Noch erstaunlicher ist das folgende Programm. Es wandelt eingegebene Großbuchstaben in Kleinschrift um. Sonderzeichen und Kleinbuchstaben werden so ausgegeben, wie sie vorher eingegeben wurden:

Die Entwicklungssystemsversion des DIGITAL 'C' führt die Umwandlung von Zeichenvariablen **nicht** vollständig aus. Daher läuft das Programm mit diesem Compiler nicht. Auf allen anderen Standardcompilern funktioniert die Umwandlung wie angegeben.

```
/* Umwandlung Großschrift-Kleinschrift */
#include "stdio.h"
#define putchar(c) putc(c,stdout)
main()
{

      int zeichen;

      scanf("%d", &zeichen);

          if (zeichen >= 'A' && zeichen <= 'Z')
              printf("c\n",zeichen + 'a' - 'A');

          else
              printf("%c\n", zeichen);

      gemdos(0x1);

  }
```

Es fällt sofort auf, daß die Character-Variable "zeichen" als "Integer" (!) deklariert wurde.

An diesem Beispiel wird deutlich, wie eine Zahlenvariable in eine Zeichenvariable umgewandelt wird. Die Integer-Variable wird in der "scanf()"-Anweisung automatisch mit dem Wert des eingelesenen Zeichens belegt. Durch den "putchar()"-Befehl wird dann das diesem Wert entsprechende Zeichen ausgegeben.

Bei der Umwandlung von Zeichen in numerische Zahlenwerte muß man jedoch darauf achten, ob dabei eine negative Zahl entsteht.

'C' legt nicht fest, ob Character-Variablen ein Vorzeichen enthalten. Entstünde bei einer Umwandlung also beispielsweise der Wert -30, so setzt der Compiler diesen auf den positiven Wert von +30.

Übrigens ist es unbedingt notwendig, bei allen "getchar()"-Anweisungen in einem Programm die eingelesene Variable wie in diesem Beispiel als **"Integer"**- und nicht als **"Character"**-Variable zu deklarieren.

Der Grund dafür liegt darin, daß "getchar()" für alle möglichen Eingaben verwendet werden kann. Zusätzlich ist ein separater Wert für das EOF, das Eingabeende, erforderlich.

Bei diesem Zeichen handelt es sich um eine Zahl. Daher kann der Resultatwert von "getchar()" nur als Zahl ausgegeben werden. Allerdings führt auch eine Character-Variable in den meisten Fällen diese Umwandlung durch.

Halten wir an dieser Stelle einmal kurz als Fazit fest:

"Character"- und **"Integer"**-Werte können beliebig kombiniert zusammen in arithmetischen Ausdrücken verwendet werden. In BASIC ist dies in dieser Form nicht möglich.

Diese Eigenschaft von 'C' ermöglicht eine sehr große Flexibilität, vor allem bei der Programmierung von **Zeichen-Transformationen**.

5.4.2 UMWANDLUNGEN UNTERSCHIEDLICHER ZAHLENTYPEN

Verwendet man unterschiedliche Zahlentypen in einer arithmetischen Anweisung, so wandelt der Compiler diese automatisch in ein bestimmtes einheitliches Zahlenformat um.

Setzen Sie beispielsweise die Anweisung

```
float_wert = integer + float ;
```

in einem Programm ein, weisen also der Variablen "float_wert" vom Typ Float eine Integer- und eine Float-Variable zu, so wird die Integer-Variable automatisch in eine Float-Variable umgewandelt.

Wie in diesem Beispiel finden generell nur Umwandlungen statt, die sinnvoll sind.

Dabei werden diese Grundregeln befolgt:

Ausgangsvariablentyp:	Umwandlung in:
short int	int
float	double

Ist danach einer der Operanden vom Typ

(Variablentyp)	, so wird der andere Operand auch in den Variablentyp umgewandelt:
double	double
long	long
unsigned	unsigned

Falls **keiner** dieser Fälle zutrift, legt der Compiler alle Operanden auf den Typ **"Integer"** fest.

Setzt man einen Float-Wert gleich einem Integer-Wert, wie in dem Beispiel

```
float_var = int_var
```

so findet auch eine Umwandlung statt. Hier wird der Bruch-Teil der "float_var" unterdrückt.

Umwandlungen können auch wieder **rückgängig** gemacht werden. Wandelt man beispielsweise eine Character-Variable in eine Integer-Variable um und anschließend wieder zurück in die Character-Variable, so erfolgt keine Änderung.

Ein Demonstrationsprogramm dazu kann so aussehen:

```
main()
{

        char zeichen;
        int zahl;

           zeichen = 'A';
           zahl = 22;

           zeichen = zahl;
           zahl = zeichen;

}
```

Der Wert der Variablen "zeichen" bleibt hierbei ungeändert, da die Änderung wieder rückgängig gemacht wurde. Die Vorabversion des DIGITAL 'C' des Entwicklungs-systems führt die Umwandlungen von Charactern jedoch **nicht** ganz vollständig aus.

Alle in diesem Abschnitt vorgestellten Umwandlungsmöglichkeiten eröffnen Ihnen als Programmierer eine bei BASIC unbekannte große Flexibilität bei der Entwicklung von Programmen. Anfangs kann sich die Vielfalt an Möglichkeiten zur Umwandlung jedoch auch als etwas verwirrend erweisen, mit einiger Erfahrung erhalten Sie aber schnell einen Überblick über alle Umwandlungsfunktionen.

5.5 VARIABLEN-DEKLARATIONEN

Wie wir bereits festgestellt haben, müssen alle Variablen vor deren Gebrauch deklariert, d.h. vereinbart und dadurch als ein bestimmter Variablentyp festgelegt werden.

Die **Deklaration** erfolgt normalerweise **im Kopfteil einer Funktion.** Sie kann jedoch auch aus dem Kontext, dem inhaltlichen Zusammenhang beispielsweise, mitten innerhalb einer Funktion auftreten.

Wichtig ist nur, daß die Deklaration vor der Verwendung einer Variablen vorgenommen wird.

Die in den vorherigen Abschnitten und Kapiteln vorgenommenen Vereinbarungen waren so einfach wie möglich gehalten und sahen beispielsweise folgendermaßen aus:

```
main()
{

    int a;
    int x;
    long y;
```

```
        float b;
        char c;

        ... weiteres Programm ...

}
```

Variablen können jedoch in beliebiger Weise innerhalb der Deklaration angeordnet werden. Möglich ist daher auch die folgende Vereinbarung:

```
main()
{

    int a, b, c, d,
        e, x-wert, y-wert;
    float zeichen_1,
          zeichen_2;

    ...

}
```

Diese Deklarationsversion ist kompakter und kürzer als die vorherige. Zur späteren Modifizierung einer Vereinbarung in einem Programm ist die Aufführung aller Variablen getrennt und untereinander jedoch oft vorzuziehen.

Wie bereits besonders hervorgehoben wurde, müssen Sie nach der Deklaration einer Variablen dieser unbedingt einen Wert zuweisen, bevor Sie mit ihr arbeiten können.

Diese Wertezuweisung haben wir bisher immer im Anschluß an den Deklarationsteil vorgenommen.

Dazu ein kurzes Beispiel:

```
main()
{

      int zahl;

      zahl = 123;
      ...

}
```

Es ist jedoch auch möglich, **Deklaration und Wertezuweisung zusammenzufassen.** Dies hat dann folgende Syntax:

```
main()
{

    int zahl = 123;

    ...

}
```

Welche Version Sie in Ihren Programmen einsetzen, hängt davon ab, ob Sie lieber eine übersichtliche, kompakte Version wie diese vorziehen, oder die Wertezuweisung erst lokal vor der eigentlichen Verwendung der Variablen des besseren Zusammenhangs wegen vornehmen wollen.

Soviel vorerst zur Variablendeklaration; auf die Vereinbarung ein- und mehrdimensionaler Arrays gehen wir an anderer Stelle in diesem Kapitel noch genau ein.

5.6 GLOBALE/LOKALE VARIABLEN

Bisher haben wir Variablen lediglich lokal, d.h. nur innerhalb einer einzigen Funktion, in unserem Fall der Funktion "main()", verwendet.

'C'-Programme bestehen jedoch üblicherweise aus einer Reihe von Programmfunktionsmodulen, die meistens aus der Hauptfunktion "main()" aufgerufen werden.

Hier ist es nun wichtig, ob Sie eine Variable für alle oder mehrere Funktionen benötigen, d.h. **global** einsetzen, oder nur lokal für Berechnungen inerhalb einer Subroutine.

Lokale Variablen, auch **automatische** Variablen genannt, werden in der Funktion deklariert, in der sie verwendet werden sollen. Sie gelten dann auch nur für dieses Unterprogramm und nicht etwa wie in BASIC für alle Funktionen.

BASIC kennt nur **globale** Variablen, d.h. die Variablen gelten für das komplette Programm mit allen Unterprogrammen.

In 'C' werden solche **globalen**, auch **"extern"** genannten Variablen vor einer Funktion deklariert, ähnlich wie wir es bisher mit den symbolischen Konstanten gehandhabt haben.

Zur Verdeutlichung beider Variablentypen finden Sie im folgenden ein Programm, in dem sowohl "automatische" als auch "externe" Variablen sowie symbolische Konstanten deklariert werden:

```
#define KONSTANTE_ZAHL 1234
int   glob_int ;
char  char_global;
main()
{

      char lokal_char = 'A' ;
```

```
        float  float_lokal;

        ...

}

subroutine(c)
{

        char lokal_char2;

        ...

{
```

Hier noch ein wichtiger Hinweis: Vor allem für BASIC-Programmierer besteht die gefährliche Tendenz, alle Variablen als "extern" zu deklarieren, um Sie immer verwenden zu können. Dies führt besonders bei vielen Funktionen zu **sehr unübersichtlichen** Programmen!

5.7 VEKTOREN-DATENFELDER

Über den Nutzen von Arrays müssen hier wohl bei Ihren BASIC-Vorerfahrungen keine Worte verloren werden. Alle Programme, die auch nur kleinste Mengen an Daten verwalten, verwenden sie.

Bisher haben wir nur kurz angeschnitten, wie einfache eindimensionale Datenfelder definiert werden.

Dem BASIC-Befehl

```
10 DIM A%(10)
```

entspricht die 'C'-Anweisung

```
int c[10];
```

Wir deklarieren Felder also genau wie alle anderen Datentypen, d.h. zunächst wird der Variablentyp definiert und anschließend ein Feldname festgelegt.

Alle Datentypen von 'C' können als **Felder** deklariert werden.

Dann wird jedoch die Feldtiefe, die Anzahl der Feldelemente, in eckigen Klammern als Index hinzugefügt.

Nach der Vereinbarung müssen, und das dürfen Sie nicht vergessen, alle einzelnen Elemente des Feldes mit einem Wert belegt werden.

Diese Wertezuweisung übergibt man sinnvollerweise einer "for"-Schleife wie in diesem Beispielprogramm:

```
main()
{

    int a[10];
    int i;

        for (i = 0; i <=10; i = i +1)
            a[i] = 0;

       ...

}
```

An der Initialisierungs-"for"-Schleife wird auch deutlich, daß Datenfelder in 'C' mit dem Element Null beginnen, nicht mit dem Element Eins, wie in BASIC.

Ein Feld mit fünf Elementen

```
int a[5];
```

enthält also die Elemente

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

nicht jedoch a(5)!

Arrays werden in 'C' ähnlich wie in BASIC behandelt.

Wollen Sie beispielsweise dem zweiten Element unseres a[5] Feldes, also a[1], den Wert 12 zuweisen, so sieht das folgendermaßen aus:

```
a[1] = 12;
```

Die Wertezuweisung kann genauso wie bei der Anweisung

```
int c = 12;
```

auch direkt in der Initialisierung erfolgen. Dies gilt jedoch mit einer **Einschränkung**: für lokale, also automatische Variablen ist dies nicht möglich.

Globale oder Static-Variablen (siehe Kapitel über 'C'-Funktionen, Static-Variablen sind lokale Variablen, die ihre Werte zwischen Funktionsaufrufen behalten) werden folgendermaßen in der Deklaration initialisiert:

```
int a[7] = { 1, 6, 8, 2, 7, 1, 15 };
```

Der Compiler verarbeitet diese Anweisung von links nach rechts und weist jedem Element des Arrays "a" der Reihe nach einen Wert innerhalb der Klammern zu.

Diese Anweisung läßt sich jedoch noch einfacher gestalten:

```
int a[] = { 1, 6, 8, 2, 7, 1, 15 };
```

Die leeren eckigen Klammern nach dem Feldnamen bewirken, daß die Feldtiefe des Arrays genau auf die Anzahl der zugewiesenen Elemente gesetzt wird.

Der Compiler zählt also die Elemente und setzt diese Zahl intern automatisch in die Klammern ein.

Ich möchte jedoch noch einmal betonen, daß diese Zuweisung nicht(!) für lokale Variablen gilt.

Dazu ein Beispiel Deklaration eines Arrays als **"Global"** mit einer Wertezuweisung:

```
int feld[] = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
main()
{

   int i;

   for (i = 0; i < 11; ++i)
       printf("%d\n", feld[i];

 }
```

Dieses Programm weist dem globalen Array "feld" die Werte von Zehn bis Null zu.

Anschließend werden diese Werte in der Funktion "main()" in einer "for"-Schleife ausgegeben.

Kommen wir nun zu mehrdimensionalen Arrays.

5.7.1 MEHRDIMENSIONALE ARRAYS

In BASIC dimensionieren wir ein Feld aus 5 mal 5 Elementen mit der Anweisung

```
10 DIM A%(5,5)
```

- in 'C' sieht dies dagegen folgendermaßen aus:

```
int a[5][5];
```

Wollen wir ein bestimmtes Element dieses Feldes ansprechen, so geschieht dies über

```
a[2][4] = 12;
```

Vergessen Sie auch hier nicht, das Feld zu initialisieren, d.h. mit Ausgangswerten zu belegen. Bei lokalen Variablen können Sie dies über zwei ineinandergeschachtelte "for"-Schleifen erledigen.

Bei globalen oder Static-Variablen ist die Initialisierung auch folgendermaßen möglich:

```
static int feld[3][5] = {{ 1, 2, 3, 4, 5},
                         { 2, 3, 4, 5, 6},
                         { 4, 5, 6, 7, 8}};
```

5.7.2 STRINGS

Der Vollständigkeit halber möchte ich an dieser Stelle noch gesondert die Strings erwähnen. Strings sind, genau betrachtet, nichts anderes als eindimensionale Zeichenketten-Arrays.

Alles wesentliche für den Programmiereinsatz von Strings wurde bereits ausführlich erläutert.

Fassen wir kurz zusammen: Ein String ist eine Zeichenkette, die durch ein Null-Zeichen ("\0") beendet wird.

Der String

```
"A"
```

ist demnach also keineswegs identisch mit dem Zeichen

```
'A'
```

da im String noch das Null-Zeichen enthalten ist, um das Ende der Zeichenkette anzuzeigen. Der String besteht also aus zwei Zeichen, aus dem "A" sowie dem Marker "\0", das Zeichen dagegen besteht nur aus dem 'A'.

Man kann einen String auch global als Character-string deklarieren. Dies sieht in einem Beispielprogramm folgendermaßen aus:

```
char string_char[] = "Ich bin eine Zeichenkette";
main()
{

        printf("%s\n", string_char);
        gemdos(0x1);

}
```

Dies ist jedoch nur mit globalen oder Static-Variablen möglich. Die String-Anweisung "printf"-"%s" bewirkt dabei den Ausdruck der globalen Zeichenkette.

Für lokale Variablen hatten wir das Problem des Formulierens von Strings über den "*"-Pointer gelöst.

Das sah beispielsweise so aus:

```
main()
{

        char *string_char;
        string_char = "Ich bin eine Zeichenkette";

                printf("%s\n", string_char);

        gemdos(0x1);

}
```

Eine genaue Erklärung, warum dies so erfolgen muß und wie es funktioniert, wurde Ihnen bisher jedoch vorenthalten, vor allem um Sie nicht gleich am Anfang mit zu vielen Informationen zu überhäufen. Diese Erläuterungen folgen nun im Detail in dem nächsten Kapitel.

KAPITEL 6:
'C'-POINTER (ZEIGER)

In 'C' hängen **Pointer** und **Vektoren**, also die auf den vorherigen Seiten ausführlich erläuterten Arrays, sehr eng zusammen. Jede der beschriebenen Feld-Operationen läßt sich auch ohne weiteres über Zeiger formulieren.

Bevor wir jedoch auf den Zusammenhang genau eingehen, ist es sinnvoll, zuerst zu klären, was Zeiger eigentlich sind und was genau sie leisten.

6.1 GRUNDLAGEN

Zeiger gibt es in BASIC nicht. Nur wenn Sie sich bereits mit der Programmierung in Maschinencode befaßt haben, werden Sie sicher Zeiger zur **indirekten Adressierung** verwendet haben.

Die Zeiger sind eine wesentliche Eigenschaft von 'C', durch die sich diese Sprache von allen anderen Programmiersprachen absetzt. Sie sind der Hauptgrund dafür, daß 'C' die bevorzugte und beliebteste Systemsprache ist und vor allem in der Erstellung professioneller Programmpakete wie z.B. "LOTUS 1-2-3" immer stärkere Verbreitung findet.

Doch was ist eigentlich ein Zeiger?

"Ein Zeiger ist eine Variable, die die Adresse einer anderen Variable enthält."

Diese Definition hört sich zunächst etwas kompliziert an, doch am folgenden Beispiel wird ganz schnell deutlich, was damit gemeint ist.

Wenn Sie normale Variablennamen festlegen, ist dieser Name für den Computer nichts als ein Platzhalter für einen Wert. Variablennamen werden also anstelle von Speicheradressen im Rechner verwendet.

Eine Wertezuweisung wie

```
int wert1 = 15;
```

bewirkt, daß der Wert 15 in den Speicherplatz der Variablen "wert1" kopiert wird.

Mittels Zeigern können Sie nun direkt auf die Speicherplatz-Adressen zeigen. Dies sieht im Programm folgendermaßen aus:

```
addr_wert1 = &wert1;
```

Der Marker "&", der sogenannte Adreßoperator, zeigt dem Compiler an, daß die Variable "addr_wert1" den Wert der Adresse der Variablen "wert1" zugewiesen bekommt. Anders gesagt weist "addr_wert1" auf die Variable "wert1" hin, zeigt also auf sie. "addr_wert1" ist der Zeiger auf "wert1".

Schön und gut, nur was bringt dies, werden Sie nun sicher fragen.

Um dies zu erläutern, stelle ich einen weiteren **Zeiger-Marker** vor. Es handelt sich um das "*"-Zeichen. Wir haben diesen Pointer-Marker bereits im Zusammenhang mit Strings verwendet, er wurde jedoch bisher nicht weiter erklärt.

Der "*"-Marker bewirkt, daß eine mit diesem Marker versehene Variable den Inhalt der Speicheradresse, auf die ein Zeiger weist, zugewiesen bekommt.

Das klingt etwas umständlich, ist jedoch ganz einfach. Angenommen, die Variable "addr_wert1" ist der Zeiger auf die Variable "wert1". Mit der Zuweisung

```
wert2 = *addr_wert1;
```

wird "wert2" der Inhalt des Speicherplatzes zugewiesen, auf die der Pointer "addr_wert1" zeigt.

Lautet der Inhalt der Ausgangsvariablen "wert1" beispielsweise 15, so weist der Zeiger "addr_wert1" auf die Speicherstelle im Computer, in der dieser Wert gespeichert ist.

Mit der Zuweisung

```
wert2 = *addr_wert1;
```

wird die Zahl 15, also der Inhalt des Zeigers "addr_wert1", der Variablen "wert2" zugewiesen. Der Inhalt von "wert2" entspricht dann dem Ursprungswert von "wert1", lautet also 15.

Unser Beispielprogramm zur Verdeutlichung lautet:

```
main()
{

      int wert1, *wert2, addr_wert1:
      wert1 = 15;

      addr_wert1 = &wert1;
      wert2 = *addr_wert1;

}
```

Es entspricht von seinen Auswirkungen der direkten Wertezuweisung ohne Zeiger in dem folgenden einfacheren Programm:

```
main()
{
       int wert1, wert2;
       wert1 = 15;
       wert2 = wert1;
}
```

Demnach sind die beiden Anweisungen

```
addr_wert1 = &wert1;
wert2 = *addr_wert1;
```

identisch mit der einfachen Wertezuweisung

```
wert2 = wert1;
```

Die Vorabversion des DIGITAL 'C' führt die Zuweisung von Pointeradressen und Pointerinhalten nicht ganz korrekt aus, d.h. die Anweisung

```
&wert2 = *addr_wert1
```

wird nicht vollständig ausgeführt. Alle anderen auf dem Markt erhältlichen ST-Compiler halten sich an den üblichen Standard

Halten wir fest:

Zeiger werden durch den Marker "&" vor einer Variable eingeführt. Die so indizierte Variable enthält dann die Speicheradresse einer zugewiesenen Variable. In

```
addr_wert1 = &wert1
```

wird "addr_wert1" die Adresse der Variablen "wert1" übergeben.

Man kann den Inhalt einer Speicheradresse, auf die ein Zeiger weist, mittels des "*"-Markers erhalten.

6.2 DER NUTZEN VON POINTERN

Nun wissen Sie, um was es sich bei Zeigern handelt. Was bringt dies jedoch an **effektiven Vorteilen** für die Programmierung?

Manchmal werden Sie feststellen, daß Zeiger die einzige Möglichkeit darstellen, spezielle Berechnungen zu formulieren.

Dies wird bei Ihren ersten intensiveren 'C'-Experimenten anfangs sicher selten der Fall sein.

Generell kann man sagen, daß Zeiger zu den wesentlichen Stärken von 'C' gehören und oftmals Voraussetzung für elegante Programme sind. Gleichzeitig müssen sie jedoch mit äußerster Sorgfalt eingesetzt werden.

Eine unüberlegte Anwendung kann Programme hoffnungslos durcheinanderbringen und alle Regeln zur optimalen, strukturierten Programmierung zunichte machen. Dies liegt vor allem daran, daß es sehr schnell passieren kann, daß Zeiger "irgendwohin" zeigen.

Eine wichtige Anwendung von Zeigern liegt im Manipulieren von Daten zwischen einzelnen Funktionen. Darauf gehen wir jedoch in dem Kapitel über Funktionen gesondert ein.

Eine der Hauptanwendungen liegt in der **Arrayverwaltung**. Wie dies funktioniert, erläutern wir in dem folgenden Abschnitt:

6.3 ZEIGER UND ARRAYS

Wie bereits zu Anfang dieses Abschnitts über Pointer erwähnt wurde, hängen Zeiger und Datenfelder eng zusammen. Dies haben wir ja auch bei der Anwendung von Strings festgestellt. Dort mußten String-Variablen als Pointer der Form

```
char *zeichenkette;
zeichenkette = "Hallo, wie stehts?";
```

verwendet werden. Der Grund dafür und viele Details mehr zu Arrays und Zeigern werden wir im folgenden genau erläutern.

6.4 NUMERISCHE DATENFELDER

Definieren wir ein eindimensionales Datenfeld mit 15 Elementen

```
float x[15];
```

so erhalten wir in unserem Vektor die Elemente a(0), a(1), a(2), a(3), bis a(14).

Vereinbaren wir nun einen Zeiger "zeiger" auf einen float-Wert

```
float *zeiger;
```

Durch die Anweisung

```
zeiger = &x[0];
```

wird "zeiger" die Adresse des Elementes Null in dem x-Feld zugewiesen.
Wie wir bereits festgestellt haben, erhalten wir den Inhalt der Speicheradresse, auf die "zeiger" weist, durch folgenden Befehl:

```
wert = *zeiger;
```

Dadurch wird der Wert von "x[0]" der Variablen "wert" zugewiesen.

Wenn jedoch

```
*zeiger
```

den Inhalt des ersten Elementes, des x-Arrays, das Element "a[0]" enthält, so kann man auch die Variable

```
*(zeiger+1)
```

festlegen. Diese weist auf das nächste folgende Element des Feldes. *(zeiger+1) entspricht damit dem Wert von "a[1]".

Generell gesagt, ist "*(zeiger+i)" identisch mit "x[i]", vorausgesetzt der "zeiger" weist wie in unserem Beispiel auch auf das x-Feld.

Gehen wir noch einen Schritt weiter. Der früher verwendete Ausdruck

```
zeiger = &x[0];
```

läßt sich auch folgendermaßen formulieren:

```
zeiger = x;
```

Die Adresse des ersten Elementes eines Arrays ist demnach identisch mit dem Namen dieses Arrays. Warum?

Bei der Übersetzung eines Programms wandelt der Compiler ein Feld automatisch in die Adresse des ersten Vektorelements um. Er erzeugt also einen Zeiger, der auf dieses Anfangselement weist. Da der Name eines Arrays identisch mit der Adresse dessen ersten Datenelements ist, läßt sich die Zuweisung "zeiger = x ;" festlegen.

Noch erstaunlicher ist die Tatsache, daß der Compiler den Ausdruck

```
x[i]
```

bei der Übersetzung automatisch in die folgende Form umwandelt:

```
*(x+i)
```

Die beiden Formulierungen für das i-te Element des Beispiel-Arrays "x" sind demnach beliebig austauschbar.

```
x[i]
```

und

```
*(x+i)
```

sind **äquivalent.**

Diese Gleichheit gilt auch für den Einsatz des Adreßoperators "&":

```
(x+i)
```

ist vollständig identisch mit:

```
&x[i]
```

Anders ausgedrückt stellt "x+i" die Adresse des i-ten Elementes des x-Arrays dar.
Dies gilt auch für die Zeiger, die genau wie das x-Feld zusammen mit einem Index in eckigen Klammern verwendet werden können:

```
zeiger[i]
```

entspricht genau

```
*(zeiger+i)
```

Man kann den Zeiger genauso verwenden wie das Feld, auf das der Zeiger weist. Es ist natürlich auch logisch, denn zu jedem Array-Element gibt es einen zugehörigen Pointer.

Alle diese Zeiger sind entsprechend der aufsteigenden Anordnung der einzelnen Feldelemente in einer identischen, aufsteigenden Weise angeordnet.

Das bedeutet, daß auf das erste Element des Arrays "x" der erste Zeiger weist, auf das zweite Element der zweite Zeiger u.s.w.

Alle diese Zeiger eines Feldes lassen sich mit einem Index der Form

```
zeiger[i]
```

oder aber

```
*(zeiger+i)
```

verwenden.

Das waren nun eine ganze Reihe geballter Informationen über die Anwendung, Details sowie Hintergründe von Zeigern in einfachen numerischen Arrays.

Alle oben beschriebenen Erläuterungen wurden anhand eindimensionaler Felder beispielhaft vorgestellt. Doch lassen sich die Aussagen ohne Einschränkungen auch auf **mehrdimensionale Felder** übertragen.

Vergewissern Sie sich, daß Sie alle Details in diesem Abschnitt auch wirklich verstanden haben. Wenn nicht, gehen Sie ihn nochmals gründlich durch. Vor allem sollten Sie die Beispiele in eigenen Programmen praktisch ausführen.

Sie benötigen dieses Grundwissen, wenn wir die numerischen Arrays verlassen und uns Character-Feldern zuwenden. Alle anhand der Zahlen-Arrays gegebenen Erläuterungen zu Pointern und Feldern gelten grundsätzlich für **alle anderen Datentypen**, auch für Characters.

Doch beinhalten Character-Arrays oder Strings noch einige zusätzliche Besonderheiten, die wir im folgenden erläutern wollen.

6.5 STRINGS UND VEKTOREN

Ein String ist nichts anderes als ein eindimensionaler Character-Array. Die Zeichenkette

```
"'C' ist die Zukunftsprogrammiersprache!"
```

wird im Rechner als Array repräsentiert. Der Compiler beendet dabei das Feld mit einem Null-Zeichen ("\0").

'C' besitzt keine spezielle Funktion zur Verwaltung von Strings, vielmehr werden alle Strings als Zeichenfelder im Speicher behandelt.

Wir haben Strings bisher wie in diesem Beispiel erzeugt:

```
main()
{

      char *zeichenkette;
      zeichenkette = "ATARI ST";

      printf("%s\n", zeichenkette);
      gemdos(0x1);

}
```

Die Anweisung

```
char *zeichenkette;
```

ist absolut identisch mit

```
char zeichenkette[];
```

An diesem Beispiel wird ganz deutlich, daß die Strings in unseren bisherigen Programmen eindimensionale Arrays sind.

Die Anweisungen

```
char *string;
```

bzw.

```
char string[];
```

legen ein Character-Feld unbestimmter Länge fest. Dieses Feld wird später im Programm mit einzelnen Elementen aufgefüllt. Das sieht dann so aus:

```
string = "beliebiger Text...";
```

Hierbei wird der Variablen "string" lediglich ein Zeiger auf die einzelnen Zeichen zugewiesen. Die Zeichenkette wird dabei nicht kopiert, an der Operation sind wirklich nur die Zeiger auf die Adressen des Textes beteiligt.

Mit der Zuweisung 'string = "beliebiger Text..." ;' zeigt demnach 'string' lediglich auf die Adresse des ersten Arrayelements, in diesem Fall auf das 'b' von 'beliebig'.

Diese Tatsache ist vor allem für BASIC-Programmierer etwas befremdend, mit einiger Progrmmierpraxis werden Sie sich jedoch daran gewöhnen.

Damit wollen wir es fürs erste genug sein lassen, was Pointer betrifft. Machen Sie sich jedoch darauf gefaßt, daß Sie ihnen im gesamten Buch ständig wieder begegnen werden.

Einen wesentlichen Stellenwert nehmen Pointer auch bei Funktionen ein. Mehr darüber erfahren Sie in dem gesonderten detaillierten Kapitel über Funktionen in 'C'.

Wenden wir uns nun im nächsten Kapitel der Arithmetik zu.

KAPITEL 7:

ARITHMETISCHE OPERATOREN UND AUSDRÜCKE

Nachdem Sie das vorherige Kapitel hoffentlich gewissenhaft durchgearbeitet haben, können Sie dieses Kapitel etwas gelassener angehen.

Pointer sind nun einmal vor allem für BASIC-Umsteiger anfangs etwas kompliziert, da sie bei BASIC unbekannt sind.

Die grundlegenden arithmetischen Operationen unterscheiden sich praktisch überhaupt nicht von den Ihnen aus BASIC bekannten Ausdrücken. 'C' besitzt jedoch eine ganze Reihe von Details, die wir noch gesondert herausarbeiten werden.

Vor allem haben Sie in 'C' einige sehr leistungsfähige arithmetische Sonderfunktionen zur freien Verfügung, die in BASIC nicht existieren.

Einige dieser Unterschiede zu BASIC und verschiedene Sonderfunktionen haben wir bereits im Einsteigerkapitel kurz angeschnitten.

Auf den folgenden Seiten werden wir diese und weitere, neue arithmetische Eigenschaften von 'C' im Vergleich zu BASIC detailliert erläutern.

7.1 WAS SIND OPERATOREN?

In der BASIC-Anweisung:

```
10 PRINT 1+5/7
```

sind die Symbole "+" und "/" die Operatoren. Diese verknüpfen die Operanden, die Konstanten 1,5 und 7.

Die Verwendung von Operatoren vollzieht sich in 'C' genauso wie in BASIC.

Sehen Sie sich dazu einmal die folgende Vergleichsliste von arithmetischen BASIC- und 'C'-Befehlen an:

```
    BASIC-Anweisung:        'C'-Befehl:
-------------------------------------------------

1.  PRINT 1+5/7              printf("%f n",1 + 5 / 7);
2.  PRINT 2.2+(4*2)/7        printf("%f n",2.2 + (4.0 *
                                          2.0 ) / 7.0 );
3.  A=888*3                  a = 888 * 3;
4.  A=5-6*2+(3*3)/2          a = 5 - 6 * 2 + ( 3 * 3 ) /
                                 2 ;
5.  A=A+1                    a = a + 1;
                             oder auch  ++a;
-------------------------------------------------
```

7.2 WERTEZUWEISUNGEN

Nachdem auf der vorherigen Seite einige der Gemeinsamkeiten von BASIC und 'C' in der Arithmetik aufgezeigt worden sind, wenden wir uns nun ersten Unterschieden zu.

Der BASIC-Ausdruck

```
10 A=A+1
```

läßt sich in 'C' wie gewohnt als

```
a = a + 1;
```

formulieren. Möglich und oft besser ist jedoch die auf den ersten Blick etwas ungewöhnliche Form

```
a += 1;
```

Diese Version ist vor allem bei längeren Ausdrücken erheblich kompakter. In "a = a + 1;" wird der Ausdruck "a" jeweils auf der linken und rechten Seite des Gleichheitszeichens wiederholt.

Die Wiederholung entfällt durch den zusammengesetzten Operator in "a += 1;".

Diese Formulierungsmöglichkeit gilt für alle herkömmlichen Operatoren, also für

```
+     -     *     /
```

sowie für die später in diesem Kapitel noch detailliert beschriebenen neuen 'C'-Operatoren:

```
%     >>     <<     &     ^     |(pipe).
```

Der arithmetische Ausdruck

```
wert_1 = wert_1 * 20;
```

kann demnach kompakter auch wie folgt geschrieben werden:

```
wert_1 *= 20;
```

Halten wir kurz fest: Sind "AUS.1" und "AUS.2" Ausdrücke, so gilt:

```
                AUS.1  Operand= AUS.2 ;
  Beispiel:       a       +   =   1
```

ist vollständig äquivalent zu

```
                AUS.1 =(AUS.1) Operand (AUS.2);
  Beispiel:       a   =   a        +       1;
```

Beachten Sie hierbei die Klammern um (AUS.1) und (AUS.2) in der obigen Formulierung. Diese werden vom Computer intern gesetzt und bewirken, daß

```
z /= x + 2;
```

wirklich genau

```
z = z / ( x + 2 );
```

entspricht und nicht etwa dem Ausdruck

```
z = z / x + 2;
```

bei dem "z" durch "x" geteilt und anschließend "2" addiert wird, statt daß wie im obigen Beispiel "z" durch "(x+2)" geteilt wird.

Sie werden sich mit dieser aus BASIC unbekannten Formulierungsmöglichkeit für Zuweisungen schnell anfreunden. Ausdrücke lassen sich so erheblich kompakter und kürzer formulieren.

Ein Beispiel: Angenommen Sie haben die Form:

```
arrwt_1e2[zz[aa]] = arrwt_1e2[zz[aa]] * 22;
```

Diese läßt sich anhand des neuen Zuweisungsoperators besser formulieren als:

```
arrwt_1e2[zz[aa]] *= 22;
```

Die Vorteile dieser Zuweisungsmöglichkeit liegen auf der Hand:

Zum einen schleichen sich beim Wiederholen des Ausdrucks rechts von dem Gleichheitszeichen keine unnötigen Fehler mehr ein.

Zum anderen werden Zuweisungen so leichter verständlich, vor allem für Zweitbenutzer Ihres Programms. Man muß nicht erst lange nachprüfen, ob der Ausdruck links identisch mit dem rechts von dem Gleichheitszeichen ist.

Außerdem spart diese Formulierung bei der Kompilierung von Programmen deutlich Speicherplatz, ermöglicht also die Generierung von effizienterem Maschinencode.

7.3 DER MODULO-OPERATOR

Dieser Operator ist in BASIC nicht vorhanden. Um dort den Rest einer Division zu bestimmen, ist ein etwas aufwendiger Rechenausdruck nötig.

In 'C' läßt sich dies ganz einfach mittels des folgenden Ausdrucks durchführen:

```
rest = a % b;
```

Der Variablen "rest" wird hier der Rest der Division von "a" durch "b" zugewiesen.

Anders gesagt, der Divisions-Operator "%" in "a % b" liefert den Rest, der entsteht, wenn man die Variable "a" durch "b" teilt.

Leider müssen Sie auch hier einige Einschränkungen in Kauf nehmen. Der **"%"-Modulo-Operator** gilt nämlich **nicht** für **"float"**- und **"double"**-Werte, sondern nur für Ganzzahlen (int) oder Zeichen (char).

7.4 DIE INKREMENT- UND DEKREMENT-OPERATOREN

Nachdem Sie vorhin gelernt haben, daß sich der für Sie als BASIC-Programmierer normal aussehende Ausdruck

```
x = x + 1;
```

auch als

```
x += 1;
```

schreiben läßt, sehen Sie unten eine weitere äquivalente, jedoch noch kompaktere Formulierung mittels des Inkrement-Operators.

Sie schaut für einen BASIC-Programmierer sehr ungewöhnlich aus und lautet:

```
++x;
```

Entsprechend gibt es den Dekrement-Operator "--". Ein Beispiel für seine Anwendung ist der Ausdruck

```
--x;
```

der vollständig identisch ist mit:

```
x = x - 1;
```

Der Dekrement-Operator "--" ist das Gegenstück zu dem Inkrement-Operator "++" und verringert den Wert der Variablen, bei der er steht.

Der Inkrement-Operator dagegen erhöht den Wert der Variablen. Die Erhöhung bzw. Verringerung beträgt dabei immer konstant "eins".

Überraschend ist nun, daß die Operatoren "++" und "--" sowohl vor, als auch nach deren Operanden stehen können.

Möglich ist die Präfix-Notierung

```
--x;
```

sowie die Postfix-Notierung

```
x--;
```

Im Prinzip unterscheiden sich die beiden Versionen auf den ersten Blick nicht. In beiden Fällen wird der Wert der Variablen "x" in unserem Beispiel um Eins verringert.

Der Unterschied liegt darin, daß in

--x; "x" dekrementiert wird, bevor der resultierende Wert verwendet wird, in

x--; die Dekrementierung von "x" erst erfolgt, nachdem der Wert derVariablen verwendet wurde.

Das klingt reichlich trocken, läßt sich jedoch schnell an einem Beispiel veranschaulichen:

Angenommen, die Variable "x" besitzt den Wert "2". In der Wertezuweisung

```
y = ++x;
```

in der die Inkrementierung sofort erfolgt, wird "y" der Wert "3" (2+1) zugewiesen. Die Variable "x" hat dann ebenfalls den Wert "3".

In der Anweisung

```
y = x++;
```

besitzt die Variable "y" nach der Wertezuweisung den Wert "2", "x" wird danach jedoch um Eins erhöht, besitzt also dann den Wert "3".

Werfen wir einmal an dieser Stelle zur Veranschaulichung einen Seitenblick auf die BASIC-Versionen dieser Operatoren:

```
y = x++     entspricht dabei        10 Y=X
                                    20 X=X+1
```

und

```
y = ++x     den beiden Anweisungen  10 X=X+1
                                    20 Y=X
```

Vor allem diese Eigenschaft der Dekrement- und Inkrement-Operatoren eröffnet in Programmen vielfältige und elegante Anwendungsmöglichkeiten. Sie werden diese Operatoren in fast jedem 'C'-Programm wiederfinden.

Wenden wir uns nun den Vergleichsoperatoren in 'C' zu.

7.5 DIE VERGLEICHSOPERATOREN

In BASIC haben wir die folgenden **Vergleichsoperatoren** zur Verfügung:

```
<    ...für.. kleiner als
>    ........ größer als
<=   ........ kleiner gleich
>=   ........ größer gleich
```

Alle diese Operatoren gelten genau wie in BASIC auch in 'C'. Hier gibt es für Sie mit Ihren BASIC-Kentnissen keinerlei Umsteigeprobleme.

Kommen wir zu den **Äquivalenz-Operatoren**. In BASIC sind dies:

```
=    ...für.. gleich
```

und

```
<>   ........ ungleich.
```

Diese unterscheiden sich beide von denen in 'C'. Dort steht

```
==   ...für.. gleich
```

und

```
!=   ...für.. ungleich.
```

Trotz dieser formalen Unterschiede ist die Anwendung der Äquivalenzoperatoren in 'C' identisch zu der in BASIC. Sie müssen sich lediglich die neuen Operatoren "==" und "!=" merken.

Besonders bei der "gleich"-Anweisung kann es Ihnen aufgrund der BASIC-Gewöhnung schnell passieren, daß Sie "=" und "==" verwechseln.

An dieser Stelle möchte ich Ihnen kurz die wichtigsten Anwendungen der Vergleichs- und der Äquivalenz-Operatoren vorstellen.

Meistens werden sie im Zusammenhang mit Kontrollstrukturen eingesetzt. Diese werden wir detailliert im nächsten Kapitel erörtern.

Hier sind Vergleiche meistens mit der "if"-Abfrage verknüpft. Ein Beispiel in BASIC ist das Programm

```
10 A=2
20 IF A=2 THEN PRINT "O.K.!"
```

In 'C' wird daraus:

```
main()
{

   int a = 2;

   if(a == 2)
     {
          printf("O.K.!\n");
     }
   gemdos(0x1);

}
```

Ein weiterer Einsatzbereich sind "while"-Schleifen. Die "while"-Anweisung kann beispielsweise folgendermaßen lauten:

```
#include "stdio.h"
main()
{

   int a = 12;
```

```
    while(a >= 11)
         {
            puts("Alles klar!");
         }
    gemdos(0x1);
}
```

Noch häufiger ist der Einsatz von **Vergleichs-** und **Äquivalenz-**Operatoren in "for"-Schleifen. Hier zur Veranschaulichung ein kurzes Programm:

```
main()
{

    int x;

    for(x = 0; x <= 20; ++x );
       printf("%d\n", x);

    gemdos(0x1);

}
```

Die Schleife zählt von Null bis Zwanzig, festgelegt durch den die obere Grenze bestimmenden Vergleichsoperator "kleiner gleich", "<=".

Noch eine Anmerkung zum Vorrang der Operatoren. Die vier Vergleichsoperatoren

```
<
<=
>=
>
```

haben untereinander alle den gleichen Vorrang. Vergleicht man Sie mit den beiden Äquivalenz-Operatoren

```
==
```

und

```
!=
```

so haben die Vergleichsoperatoren höheren Vorrang, die Äquivalenzoperatoren sind untereinander jedoch gleichwertig.

7.6 LOGISCHE VERKNÜPFUNGEN

Logische Verknüpfungen in BASIC sind

```
          AND   ........ und
sowie     OR    ........ oder
```

Die entsprechenden 'C'-Ausdrücke sehen für BASIC-Programmierer etwas ungewöhnlich aus:

Es sind

```
          &&   ...für... und
sowie     !!   ......... oder
```

Trotz der unterschiedlichen Form der 'C'-, verglichen mit den BASIC-Anweisungen, ist die Anwendung in einem Programm gleich

Die BASIC-Zeile

```
10 IF A=1 AND B=2 OR B=1 THEN PRINT "ERFÜLLT!"
```

wird in 'C' zu:

```
main()
{

    int a = 1;
    int b = 2;

        if(a == 1 && b == 2 || b == 1)
            printf("Erfüllt!\n");

    gemdos(0x1);

}
```

Sie sehen, der Aufbau beider Anweisungen ist identisch. Mit Ihren BASIC-Erfahrungen werden Sie hier keinerlei Umsteige-Schwierigkeiten haben.

7.7 DER NEGATIONS-OPERATOR

Auch dieser Operator ist eine Besonderheit von 'C' und in BASIC nicht vorhanden.

Das 'C'-Programm

```
main()
{
      int wert = 0;

      if(wert == 0)
         printf("NULL!!\n");

      gemdos(0x1);
}
```

läßt sich wie folgt auch einfacher formulieren:

```
main()
{
     int wert = 0;

     if(! wert)
          printf("NULL!!\n");
     gemdos(0x1);
}
```

Sicher werden Sie nun fragen, warum man den Ausdruck

```
if(wert == 0)
```

einfach durch

```
if(!wert)
```

ersetzen kann.

Das Ausrufezeichen vor der Variablen "wert" stellt den **Negationsoperator** dar. Dieser wird wie ein Vorzeichen in Präfix-Position verwendet.

Er liefert eine logische "1", wenn die Variable, vor der er steht, gleich Null ist. Alle logischen Werte ungleich Null interpretiert der Compiler als "wahr", wie in diesem Fall die Eins. Eine logische Null dagegen wird immer als "falsch" angesehen.

Ist der Inhalt der Variablen "wert" also ungleich Null, so erzeugt der Negationsoperator eine "logische Null" für "falsch".

In diesem Fall werden die der "if"-Abfrage folgenden Befehle nicht ausgeführt. Das können Sie schnell ausprobieren, wenn Sie in dem Beispielprogramm die Intialisierung folgendermaßen ändern:

```
int wert = 2;
```

Halten wir fest:

Der Ausdruck

```
if(wert == 0)
```

ist identisch mit der kurzen Form mittels des Negationsoperators

```
if(!wert)
```

7.8 MEHRFACHZUWEISUNGEN

'C' besitzt im Gegensatz zu BASIC auch die Möglichkeit von Mehrfach-Wertezuweisungen.

Erlaubt ist beispielsweise ein Ausdruck wie

```
a = b = c = d = 15;
```

Dieser entspricht formal den folgenden vier einzelnen Wertezuweisungen

```
a = 15;
b = 15;
c = 15;
d = 15;
```

oder wahlweise auch

```
a = 15;
b = a;
c = b;
d = a;
```

Allen vier Variablen wird durch die kombinierte Wertezuweisung der gleiche Wert zugewiesen.

Außer Konstanten, wie in diesem Fall der "15", kann auch eine weitere Variable zugeordnet werden.

Ein Beispiel dazu ist die Anweisung

```
wert_1 = wert_2 = wert_3 = variabl5;
```

in der allen drei Variablen "wert_1" bis "wert_3" der Inhalt von "variabl5" zugewiesen wird.

Mehrfachzuweisungen ermöglichen im Programmiereinsatz einen kompakteren und effektiveren, weniger Speicherplatz belegenden Programmierstil, mit dem Sie sich sicher schnell anfreunden werden.

Damit haben wir alles wesentliche zu der in 'C' verwendeten Arithmetik detailliert und mit vielen Beispielen anschaulich dargestellt, erläutert. Befassen wir uns nun abschließend im folgenden Abschnitt mit den Bit-Operatoren in 'C'.

7.9 DIE BIT-OPERATOREN

Da 'C' eine Systemsprache ist, steht dem Programmierer auch eine ganze Reihe von speziellen Operatoren zur Bit-Manipulation zur Verfügung.

Diese sind in der folgenden Liste zusammengestellt:

Bit-Operator	Funktion
!	"oder"-Verknüpfung von Bits.
&	"und" -Verknüpfung von Bits.
<<	Verschiebung eines Bits nach links.
>>	Bit - Verschiebung nach rechts.
^	exclusive"oder"-Verknüpfung von Bits.
~	Bit Komplement, Bit - Vorzeichen Operator.

Alle die Bit-Operatoren werden in genau derselben Weise wie die normalen, bereits erläuterten Operatoren in arithmetischen Ausdrücken angewendet.

Sie dürfen jedoch nicht mit **"float"**- oder **"double"**- Variablen verwendet werden.

Zur Anwendung der Operatoren in arithmetischen Ausdrücken sehen Sie unten zur Veranschaulichung eine Beispielliste:

Anwendungsbeispiel:	grundlegende Funktion:
`x = y & 011;`	Setzen von Bits auf Null.
`y = y | maske;`	Setzen von Bits auf Eins.
`y<<4;`	Verschieben von y um 4 Bits nach links
`y>>2;`	... oder 2 Bits nach rechts.
`y & 077;`	Kompensieren von Bits in einen ganzzahligen Wert.

Damit wollen wir es vorerst einmal bei dieser knappen Vorstellung der Bit-Operatoren und deren Anwendung belassen.

Dieses Buch wurde speziell für Sie als BASIC-Umsteiger und 'C'-Newcommer konzipiert. Die Bit-Manipulationen sind jedoch ein Thema, an das Sie sich erst als fortgeschrittene Systemprogrammierer wagen sollten.

Eine ausführliche Abhandlung dieses sehr speziellen Aspekts von 'C' würde Sie als Umsteiger wahrscheinlich nur mehr oder weniger erschrecken.

Wenn Sie jedoch bereits Erfahrung in der Programmierung in Maschinencode besitzen, sollten Sie anhand dieser kompakten Vorstellung der Bit-Operationen diese in Ihren Programmen anwenden können.

Aber auch als reiner BASIC-Programmierer können Sie die Operatoren ruhig einmal ausprobieren.

In der Praxis benötigen Sie sie aber nur für spezielle Systemprogrammierungen, für die auch detaillierte Systemkenntnisse erforderlich sind.
Zur normalen Programmerstellung sind Bit-Manipulatoren fast gar nicht erforderlich.

Eigentlich wurde dieser Abschnitt auch nur der Vollständigkeit halber eingefügt, vor allem weil Sie in anderen 'C'-Büchern nach diesen Informationen oft vergeblich suchen.

Als Fazit können wir festhalten:

Während in BASIC Bit-Operationen durch "AND" sowie "OR" Anweisungen vorgenommen werden, sind dazu in 'C' die auf den vorherigen Seiten vorgestellten speziellen Bit-Operatoren vorhanden.

Die BASIC-Bit-Operation

```
10 X = 7 AND 1
```

wird demnach in 'C' zu

```
x = 7 & 1;
```

Wenden wir uns nun in dem folgenden Kapitel einer detaillierten Beschreibung der 'C'-Kontrollstrukturen mit allen Besonderheiten und im Vergleich zu BASIC zu.

Kapitel 8:

Kontrollstrukturen in 'C'

Kontrollstrukturen stellen den elementaren Kern einer jeden Programmiersprache dar. Sie ermöglichen eine Festzulegung, welche Operationen der Rechner wann ausführen soll. Sie definieren im wesentlichen die Reihenfolge, in der die Aktionen durchgeführt werden.

8.1 KONTROLLSTRUKTUREN IN BASIC

In BASIC haben wir die folgenden Kontrollstrukturen zur Verfügung:

```
if ... then
for ... next
```

sowie

```
on ... goto
```

Einige strukturierte BASIC-Versionen beinhalten auch noch die beiden Anweisungen

```
if ... then ... else
```

und

```
while
```

Alle diese BASIC-Strukturen finden wir in 'C' wieder, auch wenn die Syntax sich teilweise etwas von der Ihnen aus BASIC bekannten Form unterscheidet.

Zusätzlich haben Sie jedoch noch eine ganze Reihe weiterer äußerst leistungsfähiger Möglichkeiten in 'C' zur Verfügung, mit denen Sie den Programmablauf steuern können.

8.2 DIE "IF"-ANWEISUNG

Wir haben bereits in den vorherigen Kapiteln die "if"-Anweisung verwendet und dabei festgestellt, daß sich beispielsweise das BASIC-Programm

```
10 X=15
20 Y=X
30 IF X=Y THEN (führe die folgenden Befehle aus)
```

in 'C' folgendermaßen formulieren läßt:

```
main()
{
    int x, y;

    x = 15;
    y = x;

    if(x == y)
      {
         dann führe die hier stehenden Befehle
         aus.
      }
}
```

Das Schema für den "if"-Befehl in BASIC

```
IF (Ausdruck) THEN (führe Befehle aus)
```

wird demnach in 'C' zu

```
if(Ausdruck)
  {
      (dann führe Befehle aus)
  }
```

Hierbei wird jeweis vom Compiler überprüft, ob der in den Klammern stehende arithmetische (Ausdruck) wahr oder falsch ist. Wie wir bereits erörterten, setzt 'C' eine logische Null, für **falsche** und einen von Null unterschiedlichen Wert für **wahre** Ausdrücke.

Ist also der Ausdruck ungleich Null, so werden die nach der "if"-Anweisung folgenden Befehle ausgeführt, andernfalls werden sie ignoriert und übersprungen.

Mit diesem Hintergrundwissen über die Logik der "if"-Bewertung lassen sich auch Abkürzungen formulieren. Eine dieser Abkürzungen haben wir bereits im Zusammenhang mit dem Negationsoperator "|" beschrieben (vgl. Kapitel Arithmetik).

Vergegenwärtigen wir uns noch einmal:

```
if(wert == 0)
```

läßt sich mittels des Negationsoperators als

```
if(!wert)
```

kompakter und klarer formulieren.

Eine weitere deutliche Vereinfachung liegt auch in der Formulierung

```
if(wert)
```

anstelle der Ungleichheitsabfrage

```
if(wert != 0)
```

Besitzt die Variable "wert" den Inhalt "ungleich Null", so wird diese, ohne daß die "!= 0"-Abfrage benötigt wird, als ein logisches **wahr** angesehen, und die auf "if" folgenden Befehle werden ausgeführt.
Fassen wir einmal die Informationen aus allen bisherigen Kapiteln zu der "if"-Abfrage zusammen:

Innerhalb der "if"-Bedingung können alle Vergleichs- und Äquivalenz-Operatoren stehen:

```
<
>
<=
>=
==
```

sowie

```
!=
```

Die Vergleichsoperatoren wie

```
=>
```

und

```
=<
```

sind illegal und nicht erlaubt.

AND-Anweisungen werden wie bereits erläutert durch

```
&&
```

und OR-Anweisungen durch

```
||
```

dargestellt.

Dazu ein Beispiel: die BASIC-Zeile

```
10 IF A=4 OR X>=3 AND B<>6 THEN (führe aus)
```

wird in 'C' zu:

```
if(a == 4 || x >=3 && b != 6)
  {
       (dann führe die Befehle aus)
  }
```

Passen Sie jedoch auf, daß sich nicht der folgende Fehler in Ihr 'C'-Programm einschleicht:

```
if(a = 4 || x>=3 && b !=6)
  (...)
```

Hier wurde der Zuweisungsoperator "=" mit dem Vergleichsoperator "==" verwechselt. Dieser Fehler ist geradezu typisch für Programmierer, die von BASIC kommen.

Da BASIC den Unterschied zwischen Vergleichs- und Zuweisungsoperator nicht kennt, müssen Sie sehr aufpassen, wenn Sie einen Algorithmus in 'C' übersetzen.

Übertragen wir einmal das folgende BASIC-Programm in 'C';

```
10 INPUT Y%
20 IF Y%=15 THEN PRINT "DIES IST DIE LÖSUNG":
                  PRINT "DIE ZAHL 15 STIMMT!"
```

Wir erhalten:

```
main()
{

     int y;
     scanf("%d", &y);
     if(y == 15)
       {
            printf("Dies ist die Lösung\n");
```

```
            printf("Die Zahl 15 stimmt. n");
        }
      gemdos(0x1);

}
```

An diesem Beispiel wird noch einmal deutlich, daß mehrere Folgeanweisungen auf die "if"-Bedingung in geschweiften Klammern zusammengefaßt werden müssen, sollen sie alle nur auf diese Bedingung hin befolgt werden. Bei einer einzigen Anweisung können die Klammern entfallen.

Übrigens ist es auch möglich, mehrere "if"-Anweisungen miteinander zu kombinieren. Eine solche Verknüpfung, die der AND-Kombination entspricht, ist in dem folgenden 'C'-Programmausschnitt vorgestellt:

```
if(a != 4)
 if(b == 12)
   if(x > 5)
     printf("Alle Bedingungen erfüllt!");

(weitere Programmanweisungen...)
```

Dies entspricht genau der dreifachen **AND**-Verknüpfung

```
if(a != 4 && b == 12 && x > 5)
   printf("Alle Bedingungen erfüllt!");
```

8.2.1 DIE "EXIT()"-ANWEISUNG

Nehmèn wir an, Sie wollen unser vorheriges BASIC-Programm etwas modifiziert in 'C' übertragen:

```
10 INPUT Y%
20 IF Y%=15 THEN PRINT "DAS IST DIE LÖSUNG!"
30 END
```

Neu ist der END-Befehl in Zeile 30. In 'C' wird das Programm folgendermaßen formuliert:

```
main()
{

      int y;

      scanf("%d", &y);

      if(y == 15)
        printf("Das ist die Lösung!\n");

      gemdos(0x1);
      exit(0);

 }
```

Das Grundprogramm ist Ihnen von den vorherigen Seiten ja nicht unbekannt und bedarf keiner weiteren Erklärung. Es entspricht genau dem vorherigen 'C'-Beispielprogramm.

Bisher nicht bekannt ist jedoch die Anwendung der Funktion "exit()". Durch die Anweisung

```
exit(0);
```

wird sie aufgerufen und ausgeführt.
Welcher Argumentwert dabei innerhalb der Klammern steht, ist in dieser Anwendung unerheblich. Das Argument, in diesem Fall

Null (0), wird wieder an die Funktion zurückgegeben, die "exit()" aufgerufen hat. In unserem Beispiel erfolgt die Rückgabe an die "main()"-Funktion.

In dem 'C'-Programm ist die Anweisung "exit(0);" also formal mit dem END-Befehl aus BASIC vergleichbar.

Sinnvoll läßt sich die Funktion bei der Umsetzung eines BASIC-Programms wie dem folgenden einsetzen:

```
10 INPUT Y$
20 IF Y$="STOP" THEN END
25 REM
30 PRINT "HIER GEHTS WEITER..."
```

Dies wird in 'C' so gestaltet:

```
main()
{

     char *y;
     scanf("%s", y);

     if(y == "stop")
        exit(0);

     printf("Hier gehts weiter\n");
     gemdos(0x1);

}
```

Die Anweisung

```
exit(0);
```

ist in diesem Programm äußerst wichtig.

Sie beendet das Programm genau wie das "END" und sorgt so dafür, daß die folgende "printf"-Anweisung, die "Hier gehts weiter" ausgibt, nicht durchgeführt wird.

In unserem vorigen 'C'-Programm dagegen war "exit(0)" nicht obligatorisch. Das Programm endete automatisch, da keine weiteren 'C'-Befehle folgten. In dem obigen Programm hingegen muß der Abbruch erfolgen. Sehr sinnvoll läßt sich dieser "Programmabruchsbefehl" im Zusammenhang mit "goto"-Sprüngen verwenden. Mehr darüber später in diesem Kapitel.

Die DIGITAL-'C'-Version des ATARI ST stellt Ihnen noch eine weitere ähnliche Sonderfunktion zu Verfügung. Deren Syntax lautet

```
abort(0);
```

Diese Funktion verläßt das gegenwärtige Programm, indem es einen Error generiert.

8.2.2 DIE "IF"-"ELSE"-ABFRAGE

Jeder normalen "if"-Abfrage kann wahlweise auch jeweils ein "else"-Teil zugefügt werden.

Einige BASIC-Versionen erlauben das folgende Programm, das einen "ELSE"-Befehl enthält:

```
10 INPUT A%
20 IF A%<20 THEN PRINT A%*A% ELSE PRINT "DER
   NORMALE WERT IST:";A%
```

In 'C' ist dies genauso möglich:

```
main()
{

    int a;
    scanf("%d", &a);
```

```
    if(a < 20)
      printf("%d\n", a * a);

    else
      {
          printf("Der normale Wert ist: ");
          printf("%d n",\a);
      }
    gemdos(0x1);

}
```

Der "else" Teil entspricht dabei in seiner Syntax der "if"-Anweisung. Das bedeutet, nach der Anweisung folgt kein Semikolon, und sollen mehrere Operationen ausgeführt werden, so müssen diese durch geschweifte Klammern zu einem Block zusammengefaßt werden, wie in unserem Beispiel.

Allgemein läßt sich die "if"-"else"-Konstruktion in 'C' folgendermaßen darstellen:

```
if(Bedingung erfüllt)
  {
      Befehlsblock Nr.1 ausführen
  }

else
  {
      Befehlsblock Nr.2 ausführen
  }
```

8.2.3 KOMBINATIONEN VON "IF"-"ELSE"-ANWEISUNGEN

Diese Konstruktion läßt sich beliebig durch weitere "if"- und "else"-Befehle ergänzen. Dabei tritt jedoch ein Problem auf. Angenommen, Sie finden die folgende Anweisung in einem Programm:

```
if(a == 2)
if(b != 15)
  a = a * 2;
else
  a = a * 3;
```

Hier stellt sich doch die Frage, zu welchem "if" der "else"-Befehl gehört, zu der ersten oder der zweiten Bedingung?

Diese Entscheidungsfrage wurde intern so gelöst, daß der 'C'-Compiler einen "else"-Teil immer als zugehörig zu dem gerade vorangegangenen "if"-Befehl interpretiert.

In unserem Fall gehört das "else" demnach zu dem zweiten "if"-Teil, zu "if(b != 15)".

Soll das erste "if" jedoch mit dem "else" in Beziehung stehen, so müssen Sie durch zusätzliche Klammersetzung die Zugehörigkeit ändern.

In unserem Beispiel sähe das wie folgt aus:

```
 alte Version:                  neue Version:
-------------------------------------------------

 if(a == 2)                     if(a == 2)
                                  {
    if(b != 15)                      if(b != 15)
      a = a * 2;                        a = a * 2;
                                  }

    else                        else
      a = a * 3;                        a = a * 3;
```

8.2.4 "ELSE"-"IF"-KETTEN

Die Kombination von "if"- und "else"-Anweisungen wird sehr oft verwendet, daher stelle ich sie Ihnen einmal kurz gesondert vor.

Formal sieht die "else"-"if"-Konstruktion so aus:

```
if(Bedingung 1 erfüllt)
      Befehlsblock Nr.1 ausführen

      else if(Bedingung 2 erfüllt)
              Befehlsblock Nr. 2 ausführen

           ...

               else if(Bedingung n-1 erfüllt)
                       Befehlsblock Nr.n-1 aus-
                                          führen
```

```
else
    Befehlsblock Nr. n
      ausführen.
```

Der Vorteil dieser Anordnung liegt darin, daß sich eine Entscheidung unter vielen Alternativen formulieren läßt.

In der Reihenfolge 1, 2, ..., n wird eine Bedingung nach der anderen bewertet und, wenn eine erfüllt wurde, der entsprechende Befehlsblock ausgeführt. Die gesamte "else"-"if"-Kette wird anschließend verlassen.

Die letzte Anweisung

```
else
    Befehlsblock Nr. n  ausführen
```

kann auch entfallen, sie ist nicht unbedingt erforderlich.

Da sie jedoch den Fall behandelt, daß keine der vorherigen Bedingungen zutrifft, wird sie oft benutzt, um Fehler oder unerlaubte Eingaben bzw. Bedingungen zu erkennen.

Verlassen wir damit die detaillierten Ausführungen über die "if"- und "else"-Bedingungen sowie deren Kombination und erörtern nun eine weitere entscheidende Kontrollstruktur in 'C'. In den nächsten Abschnitten stelle ich zunächst die "for"-, dann die "while"-Schleife sowie die spezielle "do-while"-Anweisung vor.

8.3 "FOR"-SCHLEIFEN

Diesen Schleifentyp haben wir in unserem Einsteigerkapitel bereits mit vielen Beispielen recht ausführlich erarbeitet.

Fassen wir diese Informationen noch einmal kurz zusammen und erörtern anschließend weitere Details der "for"-Schleifen in 'C':

8.3.1 RÜCKBLICK UND ZUSAMMENFASSUNG

Wie wir feststellten, ist die 'C'-Anweisung

```
for (x = 1; x <= 20 ; x = x + 2)
    {
       zu wiederholender Schleifeninhalt
    }
```

identisch mit der aus BASIC bekannten FOR-NEXT-Schleife

```
FOR X=1 TO 20 STEP +2
    zu wiederholender Schleifeninhalt
NEXT X
```

Die Schleifenanweisung

```
for (x = 1; x <= 20; x = x + 2)
```

besteht in 'C' ähnlich wie in BASIC aus drei Teilen:

Der erste Teil, in unserem Beispiel "x = 1;" legt den Ausgangswert der Schleife fest. Übersetzen könnte man dies mit "beginne die Schleife mit dem x-Wert 1". Dieser Teil entspricht der BASIC-Anweisung "FOR X=1".

Dann folgt mit "x <= 20;" die äußere Intervallgrenze der Schleife, vergleichbar mit "TO 20" in BASIC.

Der dritte Teil, "x = x + 2", schließlich stellt die Schrittweite der Schleife dar. Das BASIC-Äquivalent dazu lautet "STEP +2".

Die Schrittweite muß in 'C' immer angegeben werden. Auch eine Schrittweite von "+1", bei der in BASIC die Anweisung

"STEP +1" entfallen kann, muß beispielsweise mit

```
x = x + 1;
```

oder kürzer mit

```
++x;
```

festgelegt werden.

Wiederholt wird bei der "for"-Schleife in 'C' immer der jeweils direkt auf die "for-Anweisung" folgende Befehl.

Wie bei der "if"-Struktur müssen Sie, sollen mehrere Anweisungen in der Schleife wiederholt werden, diese mittels geschweifter Klammern zu einem Befehlsblock zusammenfassen.

8.3.2 UNENDLICHE SCHLEIFEN

Läßt man in der "for"-Anweisung die drei Teile 'Anfangswert', 'Endwert' sowie 'Schrittweite' einfach aus, so entsteht eine sogenannte "unendliche Schleife".

Dies sieht wie folgt aus:

```
for(;;)
  {
      unendlich oft wiederholte Befehle
  }
```

Der Grund für die unendliche Wiederholung liegt darin, daß der Compiler den mittleren Parameter, der die äußere Intervallgrenze der Schleife darstellt, in der Anweisung

```
for(;  ;)
```

als Überprüfung einer Bedingung interpretiert. Anders gesagt, wird sie jeweils nur auf ein Ergebnis wahr oder falsch überprüft. Es spielt für den Compiler keine Rolle, welche Bedingung als mittlerer Parameter steht.

Weil nun ein leeres mittleres Argument binär ungleich Null ist und somit als wahr angesehen wird, wiederholt der Compiler die Schleife immer wieder.

Mit dem folgenden Programm wird beispielsweise unendlich oft der Buchstabe "H" auf dem Bildschirm ausgegeben:

```
main()
{

    for(;;)
        printf("%c", 'H');

}
```

Übrigens haben wir auch in BASIC die Möglichkeit, Endlos-Schleifen zu formulieren, und zwar über die Anweisung:

```
FOR X=Y TO Z STEP 0
```

Das "STEP 0" bewirkt eine unendliche Wiederholung der FOR-Anweisung. Unser obiges 'C'-Programm sähe in der BASIC-Version so aus:

```
10 FOR X=1 TO 2 STEP 0
20 PRINT "H"
30 NEXT X
```

Hier müssen Sie jedoch darauf achten, daß sich die untere Intervallgrenze in der FOR-Anweisung von der oberen unterscheidet.

```
10 FOR X = 1 TO 1 STEP 0
```

bewirkt keine Endlosschleife, wohl aber

```
10 FOR X = 1 TO 4 STEP 0
```

oder

```
10 FOR X = 1 TO -30 STEP 0
```

In 'C' müssen derartige Feinheiten, anders als in BASIC, nicht beachtet werden.

Die unendliche 'C'-Schleife läßt sich nur durch BREAK oder einen GOTO-Sprung verlassen. Wie dies möglich ist, darauf wird in den folgenden Abschnitten dieses Kapitels erläutert.

Da nur die mittlere Bedingung bei der unendlichen Schleife von Bedeutung ist, sie also wahr sein muß, stellt auch das folgende Programm eine solche Schleife dar:

```
#include "stdio.h"
#define putchar(c) putc(c,stdout)
main()
{

     char y;

     for(y = 'H';;)
        putchar(y);

}
```

Auch hier wird unendlich oft der Buchstabe "H" auf dem Bildschirm ausgegeben, jedoch wird die Wertezuweisung der Character-Variablen direkt in der "for"-Anweisung vorgenommen. Der mittlere Parmeter ist aber immer noch leer und wird demnach von Compiler als wahr interpretiert.

Aus diesem Grund stellt das folgende Programm auch keine Endlos-Schleife dar:

```
main()
{

   int a;

     for(a = 1; a <20; )
        {
           printf("%d\n", a * a);
           a++;
        }
     gemdos(0x1);

}
```

Es entspräche genaugenommen nicht dem BASIC-Programm

```
10 FOR A=1 TO 20
20 PRINT A*A
30 NEXT A
```

sondern vielmehr der folgenden Version mit einer Endlos-Schleife:

```
10 FOR A=1 TO 20 STEP 0
20 A=A+1
30 PRINT A*A;
40 NEXT A
```

Bei BASIC sind derartige Endlos-Schleifen fast überhaupt nicht anzutreffen, bei 'C' finden Sie im Zusammenhang mit GOTO-Sprüngen oder BREAK-Anweisungen schon mehr Verwendung in Programmen.

8.3.3 DER KOMMA-OPERATOR

Dieser spezielle 'C'-Operator läßt sich nicht nur bei den "for"-Anweisungen einsetzen, jedoch findet er hier die häufigste Verwendung.

Seine Wirkung besteht darin, daß zwei Ausdrücke, die durch ein Komma "," getrennt sind, von links nach rechts bewertet werden. Das bedeutet, daß der Datentyp sowie der Wert des Resultats einer Anweisung automatisch Datentyp und Wert des Operanden rechts vom Komma sind.

Auf diese Weise lassen sich mehrere Ausdrücke zusammenfassen. Zuerst wird dabei der normale Ausdruck links von dem Komma-Operator ausgeführt und damit verbunden anschließend der rechte Befehl.

Konkret sieht dies folgendermaßen in einer "for"-Schleife aus:

```
main()
{

     int a, c;

     for(a = 1 , c = 15; c < 30;
                a-- , c++ )
        {
            printf("%d n%d\n", a, c);
        }
     gemdos(0x1);

}
```

Sie sehen in diesem Beispiel, wie der Geltungsbereich der "for"-Schleife mittels des Komma-Operators erweitert werden kann.

Sie können ihn dabei in dem Initialisierungsteil sowie in der Schrittweitenfestlegung einsetzen:

In der Schleifenfestlegung

```
for(a = 1 , c = 15;
                    c < 30;
                          a-- , c++ )
```

werden mit dem Initialisierungsteil der "**for**"-Anweisung

```
a = 1 , c = 15;
```

gleichzeitig vier Wertezuweisungen vorgenommen.

Genauso verhält es sich mit dem Änderungsteil des Schrittweitenzählers:

```
a-- , c++
```

Diese mittels des Komma-Operators verbundenden Inkrementierungs-/Dekrementierungs-Anweisungen werden gleichzeitig ausgeführt.

8.3.4 VERSCHACHTELTE "FOR"-SCHLEIFEN

Genau wie in BASIC lassen sich auch in 'C' mehrere "for"-Schleifen ineinander verschachteln.

Ein BASIC-Beispiel ist das Programm

```
10 FOR A% =1 TO 50
20     FOR B% = 20 TO 1 STEP -1
30        PRINT A%, B%
40     NEXT B%
50 NEXT A%
```

In 'C' werden diese Schleifenkonstruktionen wie im folgenden Programm dargestellt realisiert. Dies hat genau die gleiche Wirkungen wie das obige BASIC-Programm:

```
main()
{

     int a, b;

     for(a = 1 ; a <= 50 ; ++a)
        for(b = 20 ; a >= 1 ; --b)
           printf("%d %d\n", a, b)

        gemdos(0x1);

}
```

Weil hier die zweite "for"-Anweisung lediglich aus einer Zeile besteht, können wie üblich die geschweiften Befehlsblock-Klammern entfallen.

Erweitern wir nun unser BASIC-Beispiel um zwei weitere Zeilen:

```
10 FOR A% =1 TO 50
20    FOR B% = 20 TO 1 STEP -1
30        PRINT A%, B%
40        PRINT "X^2-WERT     Y^2-WERT"
50        PRINT A%*A%, B%*B%
60    NEXT B%
70 NEXT A%
```

In 'C' werden die beiden neuen PRINT-Befehle einfach durch die Bildung eines zusammengehörigen Befehlsblocks an die zweite Schleife angefügt.

Hier zur Veranschaulichung das modifizierte Programm:

```
main()
{

     int a, b;

     for(a = 1 ; a <= 50 ; ++a)
        for(b = 20 ; b >= 1 ; --b)
           {
              printf("%d %d\n", a, b);
              printf("x^2-Wert       y^2-Wert n");
              printf("%d %d\n", a * a, b * b);
           }
       gemdos(0x1);

}
```

Die in diesem Programm angewandten Blockklammern müssen Sie auch einsetzen, wenn Sie weitere Befehle zwischen den Schleifen aufrufen wollen.

Ein entsprechendes Beispiel unseres vorherigen BASIC-Programms lautet:

```
10 FOR A% = 1 TO 50
20    FOR B% = 20 TO 1 STEP -1
30        PRINT A%, B%
40        PRINT "X^2-WERT    Y^2-WERT"
50        PRINT A%*A%, B%*B%
60    NEXT B%
70 PRINT "HIER ÄUßERE SCHLEIFE"
80 NEXT A%
```

Bei der Umsetzung in ein 'C'-Programm müssen Sie bei der Bildung von Befehlsblocks wie folgt vorgehen:

```
main()
{

    int a, b;

    for(a = 1 ; a <= 50 ; ++a)
       {
          for(b = 20 ; b >= 1 ; --b)
             {
                printf("%d %d\n", a, b);
                printf("x^2-Wert  y^2-Wert\n");
                printf("%d %d\n",a * a, b * b);
             }

          printf("Hier äußere Schleife\n");
       }
    gemdos(0x1);

}
```

Schauen Sie sich die Klammerstruktur einmal genau an: Zunächst wurde eine innere Schleife als ein Befehlsblock gebildet. Der Block

```
{
    printf("%d %d\n", a, b);
    printf("x^2-Wert   y^2-Wert\n");
    printf("%d %d\n", a * a, b * b);
}
```

wird durch die zweite "for"-Anweisung 20mal wiederholt, bis der Wert der Variablen "b" gleich "eins" ist. Anschließend wird die äußere Schleife ausgeführt.

Zunächst gibt die Anweisung

```
printf("Hier äußere Schleife\n");
```

eine entsprechende Kontrollmeldung aus. Dies geschieht insgesamt 50mal, festgelegt durch den Wert der Variablen "a". Sie wird in der ersten "for"-Bestimmung mit einer Schrittweite von eins von fünfzig erhöht.

Übrigens ist es generell effizienter, da kürzer, für eine Schrittweitenbestimmung den in diesem Beispiel verwendeten Inkrement-Operator

```
++x;
```

anstelle der alten, Ihnen aus BASIC bekannten Zuweisung

```
x = x + 1;
```

zu verwenden. Entsprechendes gilt für den Dekrement-Operator

```
--x;
```

anstelle von

```
x =x- 1;
```

Von "eins" abweichende Schrittweiten müssen jedoch wie üblich formuliert werden. Sie müssen also beispielsweise

```
x = x + 2;
```

als Ausdruck für eine Schrittweite von **"+2"** verwenden.

8.4 SCHLEIFEN MIT "WHILE"

Nachdem wir uns ausführlich mit den in 'C' am häufigsten verwendeten "for"-Schleifen befaßt haben, wenden wir uns nun den ähnlich konzipierten "while"-Konstruktionen zu.

Zu den nun folgenden BASIC-Beispielen muß angemerkt werden, daß nicht alle BASIC-Versionen WHILE-Schleifen besitzen. Vor allem bei dem einfachen BASIC preiswerter Home-Computer sucht man eine WHILE-Anweisung vergeblich. Dennoch ist diese Konstruktion beispielsweise Microsoft-Basic-Standard, und wir wollen im folgenden nicht auf BASIC-WHILE-Beispiele verzichten.

Sollten Sie bisher jedoch noch keine WHILE-Schleifen verwendet haben, so ist dies kein Grund zur Sorge. Die Syntax sowie die Verwendung derartiger Konstruktionen ist leicht zu verstehen.

Nehmen wir als Beispiel zur Erläuterung das folgende Microsoft-Basic-Programm:

```
10 REM VON 1 BIS 10 ZÄHLEN
15 :
20 WHILE A%<10
30       A%=A%+1
40       PRINT A%
50 WEND
```

Sie sehen, daß die WHILE-Anweisung einen Block bildet, der durch das WHILE eingeleitet und durch WEND, der Abkürzung für "While END", beendet wird.

Die Anweisungen innerhalb dieses Blocks werden so lange wiederholt, wie die WHILE-Bedingung, in unserem Fall

```
WHILE A%<10
```

wahr ist. WHILE läßt sich also direkt mit **"solange"** übersetzen:

"Solange" der Wert der Variablen "A" kleiner als "zehn" ist, wird ihr Anfangswert von "0" nach dem RUN-Befehl jeweils um eins erhöht und der aktuelle Wert auf dem Bildschirm ausgegeben.

Wie sieht dies nun in 'C' aus? Schauen wir uns einmal das der BASIC-Version entsprechende Beispiel an:

```
main()
{

    int a;

    a = 0;

    while (a < 10)
          {
              ++a;
              printf("%d\n", a);
          }
    gemdos(0x1);

}
```

Die Syntax des 'C'-Programms entspricht exakt der des zuvor beschriebenen BASIC-Programms:

```
while (a < 10)
```

also "solange die Variable "a" kleiner als "zehn" ist, führe die dann als Block in geschweifte Klammern gesetzten folgenden Befehle durch".

Achten Sie aber darauf, daß Sie die in der "while"-Schleife verwendeten Variablen zuvor in Ihrem 'C'-Programm initialisiert und ihnen einen Wert zugewiesen haben. In BASIC ist dies nicht erforderlich, da dort in unserem Beispiel die Variable "A" durch RUN automatisch auf den Wert Null gesetzt wurde.

In 'C' dürfen dagegen die Initialisierung

```
int a;
```

sowie die Wertezuweisung

```
a = 0;
```

auf keinen Fall fehlen. Sie können jedoch die beiden Anweisungen, wie bereits zuvor beschrieben wurde, zu einem Befehl zusammenfassen als

```
int a = 0;
```

Achten Sie auch immer darauf, die Schrittweite, in unserem Beispiel "++a;", innerhalb des "while"-Befehlsblocks unbedingt anzugeben.

8.4.1 KOMBINATION VON "FOR"- UND "WHILE"-SCHLEIFEN

Sie können die beiden Schleifentypen sowohl in BASIC als natürlich auch in 'C' beliebig miteinander verknüpfen.

Ein entsprechendes BASIC-Beispiel ist durch das folgende Programm gegeben:

```
10 WHILE A%<20
20      A%=A%+1
30      PRINT "DURCHGANG NR.";A%
40 :
50          FOR B%=1 TO 15
60              PRINT B%;
70          NEXT B%
80 WEND
```

Und hier wie üblich die entsprechende 'C' Version:

```
main()
{

    int a = 0;
    int b;

    while(a <= 20)
        {
           ++a;
           for(b = 1 ; b <= 15 ; ++b)
               printf("%d\n", b);
        }

    gemdos(0x1);

}
```

Beide, sowohl das BASIC- als auch das 'C'-Programm, füllen zwanzig Zeilen des Bildschirms jeweils mit einer Reihe von Zahlen von eins bis fünfzehn.

Sie sehen an diesem Beispiel, wie einfach sich die beiden Schleifentypen beliebig und ohne Probleme miteinander kombinieren lassen.

8.4.2 VERSCHACHTELTE "WHILE"-SCHLEIFEN

Auch "while"-Schleifen lassen sich wie die "for"-Schleifen beliebig verschachteln. Hierzu als Demonstration ein entsprechendes Programm:

Zunächst die BASIC-Version:

```
10 WHILE A%<8
20         WHILE B%<2
30             B%=B%+1
40             PRINT A%, B%
50         WEND
55 :
60   A%=A%+2
70   PRINT "ÄUßERSE SCHLEIFE"
80 WEND
```

Und hier die 'C'-Fassung:

```
main()
{

     int a, b;
     a = b = 0;

     while(a < 8)
          {
              while(b < 2)
                   {
                      ++b;
                      printf("%d %d\n",a ,b);
                   }

             b = 0;
             a = a + 2;
             printf("Äußere Schleife\n");
          }
     gemdos(0x1);

}
```

8.4.3 DIE "DO-WHILE"-ANWEISUNG

Die "do-while"-Schleife ist eine Sonderform der normalen "while"-Schleife. Bei ihr wird die "while"-Bedingung nicht am Anfang des "while"-Blocks, sondern erst am Ende abgefragt.

Ein BASIC-Programm würde dies folgendermaßen simulieren:

```
10 FOR A=1 TO 2 STEP 0
20        B=B+1
30        PRINT B
40    IF B=100 THEN END
50 NEXT A
```

BASIC besitzt nicht die Möglichkeit einer DO-WHILE- Schleife. Daher wurde diese hier im Programm simuliert, indem zunächst eine Endlosschleife generiert wurde. In dieser zählt die Variable "B" von 1 bis unendlich.

Am Ende der Schleife wird jedoch durch die simulierte DO-WHILE-Abfrage über "IF" festgelegt, daß "B" maximal den Wert "100" besitzen kann, andernfalls wird das Programm beendet.

Kommen wir nun zu der 'C'-Version einer "do-while"-Schleife. Das obige BASIC-Programm läßt sich anhand dieser einfacher folgendermaßen gestalten:

```
main()
{

     int a = 0;

     do
        {

           ++a;
           printf("%d\n", a);
```

```
        } while (a <= 100);

    gemdos(0x1);

}
```

Die "while"-Abfrage erfolgt hier erst am Ende der durch "do" eingeleiteten Befehlsblockanweisung. Solange die "while"-Bedingung wahr ist, werden die "do"-Befehle wiederholt, ist sie falsch, verläßt der Compiler die Schleife und geht zu den nächsten Befehlen über.

Unsere BASIC-Version gilt strenggenommen auch nur für diesen Sonderfall eines Programmendes. Würden weitere Befehle auf die Schleife folgen, so würden diese nach Beendigung der "while"-Schleife noch ausgeführt werden, bei unserem BASIC-Programm aufgrund des END jedoch nicht.
Dies führt uns direkt zum nächsten 'C'-Befehl der verfügbaren Kontrollstrukturen, der "break"-Anweisung.

8.5 "BREAK" ZUM VERLASSEN VON SCHLEIFEN

Angenommen, wir erstellen folgendes kurze 'C'-Programm:

```
main()
{

    int b;

    for(b = 1 ;;)
       {
            ++b;
```

```
            printf("%d\n", b);
        }
    gemdos(0x1);

}
```

Wie bereits im Abschnitt über die "for"-Anweisungen behandelt wurde, haben wir es hier mit einer Endlosschleife zu tun. Das Programm zählt die Variable "b" von eins ausgehend um jeweils einen Wert aufwärts, ohne diesen Vorgang zu beenden.

Wie kann man derartige Schleifen verlassen? Sie haben es erraten, dies wird durch die "break"-Anweisung bewerkstelligt.

"break" wird sehr einfach in unser Beispiel implementiert.

Schauen Sie sich dazu das modifizierte Programm an:

```
main()
{

    int b;

    for(b = 1 ;;)
       {
            ++b;
            printf("%d\n", b);
            if(b == 100)
                break;
       }

    printf("Schleife abgebrochen\n");
    gemdos(0x1);

}
```

Dieses durch eine "if"-und "break"-Anweisung ergänzte Programm zählt nur bis einhundert. Ist dieser Wert erreicht, so wird die Schleife, durch "break" bedingt, abgebrochen und verlassen.

Der Compiler fährt anschließend mit der Abarbeitung der im Programm folgenden Befehle fort. In diesem Fall wird die Anweisung

```
printf("Schleife abgebrochen\n");
```

ausgeführt.

"break" entspricht also nicht einem "END" in BASIC, sondern dient zum Verlassen von

for-
while-

sowie

do-Schleifen.

In BASIC könnte man unser 'C'-Programm folgendermaßen darstellen:

```
10 FOR A=1 TO 2 STEP 0
20     B=B+1
30     PRINT B
40        IF B=100 THEN GOTO 60
50 NEXT A
55 :
60 PRINT "SCHLEIFE ABGEBROCHEN"
```

Hier wird deutlich, daß das Programm durch die ein "break" simulierende GOTO-Anweisung in Zeile 40 anstelle eines END nicht beendet wird.

Lediglich die Endlosschleife wird verlassen und der Programmablauf in Zeile 60 weiter fortgeführt.

8.6 DIE "CONTINUE"-ANWEISUNG

Die "continue"-Anweisung ist das genaue **Gegenstück** zu dem "**break**"-Befehl.

Sie sorgt dafür, daß die nächste Wiederholung der umgebenden

```
for-
while-
```

oder

```
do-Schleife
```

unmittelbar begonnen wird.

Das bedeutet: Gelangt der Compiler in einem Programm innerhalb einer Schleife an die "continue"-Anweisung, so werden die in dieser Schleife nach "continue" stehenden Befehle nicht mehr bearbeitet, sondern statt dessen der nächste Schleifendurchgang umgehend begonnen.

Veranschaulichen wir dies an einem 'C'-Programm:

```
main()
{

     int a[5];
     int i;

     a[0] = 15;
     a[1] = -2;
     a[2] = 0;
     a[3] = 12;
     a[4] = -14;

     for(i = 0 ; i <= 4 ; ++i)
        {
             if(a[i] <= 0)
               {
                   printf("Wert ist negativ!\n");
```

```
                continue;
            }

            printf("Wert ist positiv!\n");
        }
    gemdos(0x1);

}
```

Zunächst wurde ein Array mit fünf Elementen definiert, anschließend jedem einzelnen Element jeweils ein positiver oder negativer Integer-Wert zugewiesen.

In der dann folgenden "for"-Schleife werden die Elemente bewertet, je nachdem, ob sie positiv oder negativ sind.

Ist ein Array-Element negativ, so muß nach der Ausführung des Befehls

```
printf("Der Wert ist negativ!\n");
```

nicht mehr überprüft werden, ob das Element positiv ist. Es kann also sofort das nächste Element des Feldes untersucht werden. Mit der Anweisung

```
continue;
```

wird bewirkt, daß unmittelbar die nächste Wiederholung der "for"-Schleife begonnen wird.

Die nach "continue;" folgenden Befehle oder in unserem Beispiel der Befehl

```
printf("Der Wert ist positiv!\n");
```

werden nicht mehr berücksichtigt, was ja auch sinnvoll ist.

In BASIC entspricht die direkte NEXT-Anweisung etwa dem "continue"-Befehl.

Vergleichen Sie dazu das 'C'-Programm mit der entsprechenden BASIC-Version:

```
10 DIM A(5)
20 :
30 FOR I = 1 TO 5 STEP 1
40 :    IF A(I) <= 0 THEN PRINT "WERT IST NEGATIV!":
                          NEXT A:    REM "CONTINUE"
50 :    PRINT "WERT IST POSITIV!"
60 NEXT A
```

8.7 DER "GOTO"-SPRUNG

Auch 'C' verfügt über eine **"goto"**-Anweisung, die sogar noch etwas flexibler als die aus BASIC bekannte Strukturen ist.

In BASIC basiert ein Großteil der Kontrollstruktur auf GOTO-Befehlen. Die massive Anwendung dieser kann jedoch schnell zu unübersichtlichen "Spagetti"-Programmen führen, in denen man sich vor lauter GOTO-Befehlen nicht mehr zurechtfindet.

Um diese Gefahr von vornherein auszuschließen, wurden in diesem Buch bei allen Beispielen keine derartigen "goto"-Sprünge verwendet. Sie werden die "goto"-Anweisung auch sehr selten in 'C'-Programmen finden und wenn, dann nur vereinzelt.

Auch Dennis Ritchie, der Entwickler der Sprache 'C', empfiehlt mit Nachdruck, die "goto"-Marken möglichst gar nicht, oder wenn überhaupt, **so sparsam wie möglich**, einzusetzen.

Daran sollten auch Sie sich halten und die GOTO-Sprünge aus BASIC nicht auf 'C'-Programme übertragen.

Für die Fälle, in denen "goto"-Anweisungen in 'C' unter Umständen nützlich sein können, folgt auf den nächsten Seiten eine ausführliche Beschreibung dieser Kontrollstruktur.

Das DIGITAL 'C' des ATARI-ST-Entwicklungssystems besitzt diesen Befehl noch nicht, in der kommerziellen Endversion wird er voraussichtlich implementiert sein.

8.7.1 DIE "GOTO"-SYNTAX

Gehen wir wie gewohnt wieder von einem BASIC-Beispiel aus:

```
10 A%=10
20 PRINT A%
30 A%=A%-1
40 IF A%>0 THEN GOTO 20
```

Die Übertragung auf 'C' ergibt das folgende Programm:

```
main()
{

    int a = 10;

    20:
        printf("%d\n", a);
        --a;

        if(a > 0)
          goto 20;
        gemdos(0x1);

}
```

Beide Programme sind in ihrem Aufbau identisch und führen einen Count-Down von 10 bis 1 durch.

Sie sehen, daß "goto" in diesem Beispiel genau wie in BASIC angewendet wurde.

Der Befehl

```
goto 20;
```

bewirkt dabei einen direkten Sprung zu der Zeile 20, die vorher durch

```
20:
```

in unserem Programm markiert wurde.

In 'C' werden jedoch, anders als in BASIC, die "goto"-Markierungen als Namen und nicht etwa als Zeilennummern angesehen.

Sie können in unserem Programm den Marker "20" einfach durch einen Namen ersetzen.

In dem folgenden Programm wurde anstelle von "20" der Name "IBegin" für "If Begin" als Schleifenbeginn-Marker eingesetzt. Dies sieht dann so aus:

```
main()
{

     int a = 10;

     IBegin:
             printf("%d\n", a);
             --a;

             if(a > 0)
               goto IBegin;
             gemdos(0x1);

}
```

8.7.2 VERMEIDEN VON "GOTO"-SPRÜNGEN

An dem Beispiel wird gleichzeitig deutlich, warum in 'C' die "goto"-Sprünge äußerst selten sind. Unser Programm läßt sich ohne "goto", beispielsweise mittels einer **"while"**-Schleife, erheblich kürzer und effektiver formulieren:

```
main()
{

     int a = 10;

     while(a > 0)
          {
              printf("%d\n", a);
              --a;
          }
     gemdos(0x1);

}
```

Hier wird klar ersichtlich, daß "goto"-Sprünge fast immer durch eine andere Schleifenkonstruktion oder aber durch Funktionen effektiv ersetzt werden können. Man kann also jedes Programm grundsätzlich auch ohne **"goto"** formulieren.

Wo könnte aber ein "goto"-Sprung wirklich sinnvoll sein?

8.7.3 "GOTO"-ANWENDUNGSMÖGLICHKEITEN

In der Praxis zeigt sich, daß die häufigste Anwendung dieses Sprungs in dem speziellen Fall des Verlassens mehrerer Schleifen liegt.

Muß beispielsweise bei einem schwerwiegenden Fehler aus einer tief geschachtelten Schleifenkonstruktion herausgesprungen werden, so ist "goto" sehr hilfreich.

Man könnte zwar auch "break" verwenden, bedenken Sie aber, daß "break" lediglich eine einzige Schleife verläßt. Um aus einer Reihe von verschachtelten Schleifen herauszuspringen, ist "goto" effektiver, da "break" in diesem Fall mehrmals hintereinander ausgeführt werden müßte.

Bei der Anwendung von "goto" ist noch zu beachten, daß ein Sprung jeweils nur zu einem Punkt innerhalb einer Funktion erfolgen kann.

Belassen wir es hierbei und wenden uns einer wesentlich häufiger verwendeten Sprungkonstruktion zu, der "switch"-Anweisung.

8.8 BEDINGTE AUSFÜHRUNG DURCH "SWITCH"

Die **"switch"**-Anweisung untersucht einen Ausdruck auf einen oder mehrere konstante Werte. Sie stellt damit eine besondere Auswahl mehrerer Verzweigungsmöglichkeiten dar.

In BASIC entspricht diese Kontrollstruktur in etwa der ON___GOTO- oder ON___GOSUB- Anweisung. Diese beiden Verzweigungsarten sind allerdings erheblich unflexibler als die "switch"-Konstruktion in 'C'.

Veranschaulichen wir dies wieder an einem Beispiel.

8.8.1 BEISPIEL

Zunächst das BASIC-Programm:

```
10 INPUT A
20 ON A GOTO 40,50
30 PRINT "DEFAULT, WERT <>1 ODER 2"
40 PRINT "A IST 1":END
50 PRINT "A IST 2":END
```

Anschließend die 'C'-Version, realisiert mittels der "switch"-Anweisung:

```
main()
{

      int a;
      scanf("%d", &a);

      switch(a)
            {

                  case 1:
                        printf("Wert ist 1\n");
                        break;
                  case 2:
                        printf("Wert ist 2\n");
                        break;
                  default:
                        printf("default,Wert!=
                                    1 oder 2\n");

            }
      gemdos(0x1);

}
```

Die beiden Programme fragen eine Zahl über die Tastatur ab und bestimmen dann, ob diese "1", "2" oder ungleich diesen beiden Werten ist.

In 'C' lautet das Darstellungsformat der "**switch**"-Anweisung

```
switch(x)     (Schalter)
case  y:      (Fall)
default       (Standardfall)
```

8.8.2 DIE "SWITCH"-SYNTAX

Eröffnet wird die "switch"-Kontrollstruktur mit dem Befehl

```
switch(a)
```

Dies legt die zu bewertende Variable fest. Achten Sie darauf, daß diese Variable oder der entsprechende Ausdruck in den runden Klammern immer einen Integerwert darstellen muß.

Die nun folgenden Fallzuweisungen werden durch geschweifte Klammern zu einem Befehlsblock zusammengefaßt.

Die einzelnen "case"-Verzweigungsfälle werden dann beispielsweise durch die Anweisung

```
case 1:
        printf("Wert ist 1\n");
```

bestimmt. Ist der Wert der Variablen "a" gleich "eins", so werden die auf "case 1:" folgenden Operationen ausgeführt. Mehrere Anweisungen müssen dabei ausnahmsweise nicht als Befehlsblock durch geschweifte Klammern zusammengefaßt werden.

Liegt keine "case"-Konstante vor, so wird mit der

```
default:
```

Marke fortgefahren. Die nach "default" folgenden Anweisungen werden dabei ausgeführt. Default ist optional, sie müssen es nicht verwenden.

Existiert keine "default"-Marke in Ihrem Programm und wurde keine zutreffende "case"-Bedingung ausgewählt, so findet in der "switch"-Anweisung keine Aktion statt.

Übrigens müssen die einzelnen "case"-Bedingungen, anders als in BASIC, nicht in numerischer Reihenfolge (1, 2, ... default) angeordnet sein. Möglich ist auch das folgende Beispiel mit der ungeordneten Reihenfolge 2, 3, 1 und default:

```
main()
{

     int a;
     scanf("%d", &a);

     switch(a)
          {

               case 2:
                    printf("Wert ist 2\n");
                    break;
               case 3:
                    printf("Wert ist 3\n");
                    break;
               case 1:
                    puts("Wert ist 1");
                    break
               default:
```

```
                    printf("Wert!=1, 2
                    oder 3\n");
            }
        gemdos(0x1);

    }
```

Die "switch"-Anweisung ist nicht nur auf Zahlen, wie in den bisherigen Beispielen, begrenzt, sondern gilt auch für Character-Werte.

Dies wird in dem folgenden Programm demonstriert:

```
main()
{

    char z;
    z = 'h';

    switch(z)
        {

            case 'y':
                printf("Character y\n");
                break;
            case 'a':
                printf("Character a\n");
                break;
            case 'h':
                printf("Character h\n");

        }
    gemdos(0x1);

}
```

Im Gegensatz zu den vorigen "switch"-Programmen werden die "case"-Anweisungen als

```
case 'Character':
```

formuliert, anstelle von:

```
case Numerischer Wert:
```

Eine weitere besondere Funktion von "switch" im Gegensatz zu dem ON-GOTO des BASIC ist die Möglichkeit der **Mehrfachzuweisungen.**

Das folgende Programm stellt mittels "case"-Bedingungen fest, ob ein Character eine Zahl oder ein Sonderzeichen darstellt.

Dies läßt sich auch ohne "case" über die ASCII-Bestimmung der Eingabe (vgl. Kapitel über Bildschirm-Input-/Output-Operationen) effektiver gestalten, doch dient das Programm hier zur Veranschaulichung dieser Möglichkeit der "switch"-Anweisung:

```
main()
{

    char a;
    a = '6'

    switch(a)
        {
            case '0':
            case '1':
            case '2':
            case '3':
            case '4':
            case '5':
            case '6':
            case '7':
            case '8':
            case '9':
```

```
                printf("Character ist eine
                        Zahl!\n");
                break;

        default:
                printf("Character ist ein
                        Buchstabe oder ein
                        Sonderzeichen\n");
    }
 gemdos(0x1);
}
```

Von einer Alternative wird in diesem Programm unmittelbar zur nächsten übergegangen. Es können also mehrere "case"-Marken vor einer einzelnen Anweisung stehen.

Allgemein läßt sich dieser Fall so skizzieren:

```
switch(x)
     {
        case a:
        case b:
        case c:
               Bedingung ausführen, wenn
               a,b oder c erfüllt, d.h. wahr
               sind.
     }
```

Beachten Sie in unserem Beispiel auch die Anwendung der "break"-Anweisung. Sie findet in der "switch"-Struktur eine sehr starke Verwendung.

Ist bereits ein Fall zugewiesen worden, also eine "case"-Bedingung wahr, so ist es ja wenig sinnvoll, alle anderen "case"-Abfragen zu überprüfen.

Statt dessen kann die "switch"-Anweisung sofort über "break" verlassen werden. Auch unsere vorherigen Beispiele können alle durch "break" optimiert werden.

Zusätzlich ein Hinweis, der sich eigentlich erübrigt. Alle "case"-Konstanten in einer "switch"-Anweisung müssen aus Gründen der Logik voneinander verschieden sein.

Damit ist unser Kapitel über die in 'C' verfügbaren Kontrollstrukturen abgeschlossen. Eine ganze Reihe von Strukturen ist Ihnen bereits von BASIC her bekannt. Das Umsteigen auf die entsprechenden 'C'-Anweisungen sollte Ihnen aufgrund der vielen anschaulichen BASIC-Beispiele keine größeren Umstellungsschwierigkeiten bereiten.

Für die zusätzlichen Möglichkeiten, die detailliert dargestellt wurden und über die BASIC nicht verfügt, gilt entsprechendes.

Experimentieren Sie mit allen vorhandenden Befehlen und genießen Sie die aus BASIC nicht bekannte Flexibilität der 'C'-Strukturen.

Kapitel 9:

Charakteristische Fehler von BASIC-Umsteigern

Nun sind Sie an einem Punkt angelangt, an dem Sie bereits einen großen Teil von 'C' beherrschen sollten. Vor allem müßte Ihnen die Beziehung zwischen BASIC und 'C' klar geworden sein.

Bei dem bisher vermittelten Wissen handelt es sich um die ausführlich dargestellten Basiselemente der Sprache 'C'.

Bevor wir uns in den folgenden Kapiteln mit den weiterführenden Strukturen ausführlich befassen, wollen wir uns hier die häufigsten Fehler vor Augen führen, mit denen erfahrungsgemäß viele BASIC-Umsteiger konfrontiert werden.

Die gleich vorgestellten Programmfehler helfen Ihnen, diese typischen Fehler zu vermeiden.

Betrachten Sie dieses Kapitel als "Quiz". Zunächst wird Ihnen jeweils ein Programm mit einem einzelnen Fehler vorgestellt. Dann folgt eine ausführliche Erklärung dieses Fehlers.

Versuchen Sie dabei, falls Sie die fehlerhafte Anweisung nicht sofort finden sollten, nicht gleich die Erklärung zu überfliegen; suchen Sie gründlich, manche Fehler werden erst auf den zweiten Blick ersichtlich.

Viel Erfolg bei der Fehlersuche!

9.1 FEHLER-PROBLEM NR.1

```
main()
{

    int x;
    x = 15;

    print("%d", x);
    gemdos(0x1);

}
```

Haben Sie den Fehler gefunden? Nein? Auf den ersten Blick scheint das Programm auch völlig in Ordnung zu sein. Dies ist ein typischer Fehler, der wirklich nur einem BASIC-Umsteiger passieren kann. Jemand, der 'C' als erste Programmiersprache lernt, wird einen derartigen Fehler sicher nicht machen.

Falls Sie es nicht bemerkt haben sollten, die "print()"-Anweisung ist zwar in BASIC von PRINT her bekannt, doch die korrekte 'C'-Anweisung lautet "printf()" und nicht "print()".

Ein derartiger Gewohnheitsfehler ist ganz besonders frustrierend, da er nur schwer zu finden ist.

9.2 FEHLER-PROBLEM NR.2

```
main()
{

    int ganzzahl;

    while(ganzzahl < 10)
        {
```

```
            printf("Wert    x^2-Wert\n");
            printf("%d %d\n",ganzzahl, ganzzahl
                                    * ganzzahl);
            ++ganzzahl;
        }
    gemdos(0x1);

}
```

Das Programm bildet in seiner **"while"**-Schleife eine Wertetabelle mit den Quadratwerten der Variable **"ganzzahl"** von null bis zehn.

Der hier dargestellte Fehler ist genau wie der vorherige ein ganz spezifischer Umsteiger-Fehler.

Schauen Sie sich einmal genau die Initialisierung des Programms an:

Nach der Anweisung

```
int ganzzahl;
```

muß vor der **"while"**-Schleife unbedingt die Wertezuweisung

```
ganzzahl = 0;
```

erfolgen.

In BASIC ist dies nicht erforderlich, da der Wert aller Variablen durch **RUN** automatisch auf Null gesetzt wird. In 'C' dagegen dürfen Sie nie vergessen, die Ausgangswerte festzulegen.

Dies kann, wie bekannt, auch durch die kombinierte Initialisierungs-/Wertezuweisungs-Anweisung

```
int ganzzahl = 0;
```

erfolgen.

9.3 FEHLER-PROBLEM NR.3

```
main()
{

     int ganzzahl;

     for(ganzzahl = 0 ; ganzzahl <= 10 ;
         ++ganzzahl);
        {
            printf("Wert    x^2-Wert\n");
            printf("%d %d\n", ganzzahl,
                     ganzzahl * ganzzahl );
        }
     gemdos(0x1);

}
```

Dieses Programm entspricht exakt dem vorherigen Beispiel, nur wird hier die Wiederholung durch eine "for"- anstelle einer "while"-Schleifenkonstruktion bewirkt.

Haben Sie den Fehler noch nicht gefunden? Sie können auch keinen finden, dieses Programm ist fehlerfrei.

Man könnte annehmen, daß wie in dem letzten Beispiel die Wertezuweisung der Variablen **"ganzzahl"** fehlt. Doch diese Initialisierung wird in der Schleifenanweisung vorgenommen, muß also nicht gesondert vorher im Programm erfolgen.

9.4 FEHLER-PROBLEM NR.4

```
main()
{

      int wert;

      scanf("%d", &wert);

      if(wert = 15)
        printf("15 ist die Lösung\n");

      else
        printf("Das war nichts!\n");

      gemdos(0x1);

}
```

Was, in diesem Programm soll ein Fehler sein? Ich nehme an, daß Sie dies gerade gedacht haben. Für einen BASIC-Umsteiger sieht es völlig normal aus.

Und doch ist es fehlerhaft! Schauen Sie sich einmal die "if"-Bedingung genauer an. In den vorangegangenen Kapiteln wurde immer wieder betont, daß Wertezuweisungs-Gleichheitszeichen "=" nicht mit dem Vergleichs-Gleichheitszeichen "==" zu verwechseln. Die korrekte "if"-Bedingung muß natürlich "if(wert == 15)" lauten und nicht etwa "if(wert = 15)".

9.5 FEHLER-PROBLEM NR.5

```
main()
{

    int wert_1, wert_2;

    wert_1 = 15;
    wert_2 = 3.5;

    printf("%d\n", wert_1 * wert_2);
    gemdos(0x1);

}
```

Hier werden zunächst die beiden Variablen "wert_1" und "wert_2" definiert, diesen dann Werte zugewiesen und in der "printf"-Anweisung der Quotient beider Variablen ausgegeben.

Haben Sie bemerkt, wo hier der Fehler liegt?

Der Fehler steckt in den beiden Wertezuweisungen. Da die Variable "wert_2" genau wie "wert_1" als Integer, also Ganzzahl definiert wurde, kann man ihr nicht den Gleitzahlenwert "3.5" zuweisen.

Dieses Programm wird dennoch von den meisten Compilern ausgeführt werden, jedoch mit der unerwünschten Nebenwirkung, daß "wert_2" der Wert "3.5" als Integer, d.h. als "3", zugewiesen wird.

Wünschen Sie jedoch einen Wert von "3.5", so müssen Sie erstens "wert_2" als "float"-Variable definieren. Zweitens muß auch die "printf"-Anweisung geändert werden. Anstelle von "%d" muß sie eine "%f"-Formatanweisung für Gleitkommazahlen enthalten.

9.6 FEHLER-PROBLEM NR.6

```
main()
{

      a = 15;

      printf("%d\n", a);
      gemdos(0x1);

}
```

Hier wird ein weiterer sehr typischer "BASIC-Fehler" begangen.

Auf den ersten Blick scheint das Programm korrekt zu sein, die **"printf"**-Anweisung ist fehlerfrei, die Blockklammern der Funktion **"main()"** sind richtig gesetzt. Auch die Wertezuweisung der Variablen **"a"** ist in Ordnung.

Dennoch versteckt sich hier ein Fehler. In BASIC ist es nicht erforderlich, eine Variable zu definieren, in 'C' dürfen Sie dies auf gar keinen Fall vergessen.

Fügen Sie also die Anweisung

```
int a;
```

oder

```
float a;
```

nachträglich am Anfang des Programms ein, je nachdem, ob die Variable "a" als Ganzzahl oder Gleitzahl definiert werden soll.

9.7 FEHLER-PROBLEM NR.7

```
main()
{

     printf("%f\n", 1 / 3);
     gemdos(0x1);

}
```

Das Programm soll falsch sein? Es stimmt wirklich!

Wenn Sie es nicht glauben, können Sie es ja in Ihren 'C'-Compiler eingeben - er wird entweder das fehlerhafte Ergebnis

```
0.000000
```

ausgeben oder einen Error generieren.

Der Fehler kann nur in der "printf()"-Anweisung liegen. Diese soll den Bruch 1/3 ausgeben. Das Ausgabeformat wurde auch korrekt als "%f", also als "float"-Wert festgelegt.

Wie in dem Kapitel über Bildschirm-Ein- und Ausgabe- Operationen bereits betont wurde, müssen dann auch die beiden Zahlen "1" und "3" als "float"-Werte, nicht aber als "integer", wie in diesem Fall, gewählt werden.

Das fehlerfreie Programm lautet demnach:

```
main()
{
     printf("%f\n", 1.0 / 3.0);
     gemdos(0x1);
}
```

9.8 FEHLER-PROBLEM NR.8

```
main()

    int a, b, c, d;

    scanf("%d", &a);

    b = a + 6;
    c = b * 4;
    d = b - c;

    printf("%d %d %d %d\n", a, b, c, d);
    gemdos(0x1);

}
```

Der Fehler in diesem Programm wird vielen von Ihnen sicher sofort ins Auge gefallen sein. Das Problem besteht hier darin, daß der Compiler eine von mehreren unterschiedlichen Fehlermeldungen ausgibt. Keine von diesen gibt jedoch den genauen Grund des Fehlers an.

In diesem Beispiel wurde die erste geschweifte Block-Klammer nach der "main()"-Funktionsdefinition vergessen. In dem kurzen Demo-Beispiel ist der Fehler schnell gefunden, doch in einem auch nur etwas umfangreicheren 'C'-Programm ist die Suche aufgrund der unbestimmten Fehlermeldung äußerst schwierig.

Denken Sie daran, daß umfangreichere Programme nicht nur einen einzigen Befehlsblock wie in diesem Beispiel, sondern eine ganze Reihe von diesen besitzen.

Wie Sie wissen, beinhalten beispielsweise die meisten Schleifen mehrere Blöcke.

Wenn dann in tief verschachtelten Schleifen eine Blockklammer vergessen wird, dann suchen Sie mal schön, vor allem wenn Sie nicht genau wissen, wonach Sie suchen sollen, da der Compiler Ihnen keine genaue Fehlerbeschreibung als Hilfestellung gibt.

Vergewissern Sie sich also immer, daß Sie alle Befehlblöcke jeweils mit einer geschweiften Klammer eröffnen und auch wieder schließen, vor allem bei tief verschachtelten Schleifenanweisungen.

9.9 FEHLER-PROBLEM NR.9

```
main()
{

    int schleife;

    for(schleife = 1 ; schleife =< 10;
            ++schleife );
       {
          printf("%d\n", schleife, schleife *
                              2 );
       }
   gemdos(0x1);

}
```

Zugegeben, der Fehler ist schwer zu erkennen.

Schauen Sie sich doch einmal die "for"-Schleifendefinition etwas näher an. Der Error wird hier durch die Bedingung "schleife =< 10" generiert. In einigen BASIC-Versionen ist der "=<"-Vergleich zulässig, in 'C' jedoch nicht.
Die korrekte Version muß "schleife <= 10;" lauten.

Denken Sie immer daran, daß in 'C' das Gleichheitszeichen in einer "kleiner-/größer-/gleich"-Anweisung am Ende steht, also "<=" bzw. ">=" lauten muß.

9.10 FEHLER-PROBLEM NR.10

```
main()
{

     int wert;
     scanf("%d", x);

     printf("%d %d\n", x, x * x);
     gemdos(0x1);

}
```

Auf den ersten Blick scheint sich hier kein Fehler zu verbergen ... oder?

Wie war das noch mit der "scanf"-Anweisung?

Erinnern Sie sich jetzt wieder, falls Sie den Fehler nicht sofort gefunden haben? "scanf" benötigt immer Zeigervariablen, die dem Rechner mitteilen, an welcher Speicherstelle der eingelesene Variablenwert abgelegt werden muß.

Die korrekte "scanf"-Anweisung muß demnach mit einem "&"-Zeiger formuliert als

```
scanf("%d", &x);
```

in das Programm eingefügt werden.

9.11 FEHLER-PROBLEM NR.11

```
main()
{

      int a, b, c, d;
      a = b = 15;

      for(c = a , d = c * 2 ; a < 5 ; ++a, ++d)
            {
            printf("%d\n", a);
            c = a * 12;
            b = c - a;
            printf("%d %d\n", c, b);
      gemdos(0x1);

}
```

Lassen Sie sich durch dieses Programm nicht verwirren.

Der Fehler ist diesmal ganz einfacher Natur. Er verbirgt sich auch nicht in der "for"-Anweisung, obwohl diese durch die Komma-Operatoren etwas erweitert wurde.

Sie sollten ihn, nach den zu der Klammerstruktur in dem Fehler-Problem Nr.8 gegebenen Informationen, an sich sofort finden.

Genau, hier wurde die geschweifte Endklammer des "for"-Blocks vergessen. Diese muß nach der letzten "printf"-Anweisung noch eingefügt werden.

9.12 FEHLER-PROBLEM NR.12

```
main()
{

     int a,d;
     float b;
     char *c;

     scanf("%d %f %s", &a, &b, &c);

     for(d = 1; d < a ; ++d)
        printf("%s\n", c);
     gemdos(0x1);

}
```

Dies Programm liest einen String sowie einen Integer- und einen Float-Wert über die "scanf"-Anweisung ein und gibt den String anschließend in der "for"-Schleife a-mal untereinander auf dem Bildschirm aus.

Fehlerhaft ist hierbei die "scanf"-Anweisung. Sollten Sie den Fehler noch nicht gefunden haben, so schauen Sie sich diese jetzt einmal genauer an.

Die Stringvariable "c" ist bereits als Pointer definiert worden, benötigt also kein zusätzliches Markerzeichen "&". Die Anweisung lautet korrekt:

```
scanf("%d %f %s", &a, &b, c);
```

9.13 FEHLER-PROBLEM NR.13

```
main()
{

    char *string;
    string = "||";

    printf(%s "Fehler", string);
    gemdos(0x1);

}
```

Dies Programm soll die Meldung

```
"Fehler!!"
```

auf dem Bildschirm ausgeben.

Es denkt jedoch gar nicht daran. Dies liegt an der "%s"-Anweisung innerhalb von "printf".

Diese und alle anderen Typenanweisungen müssen immer in Anführungszeichen eingebunden werden.

Die korrigierte Zeile lautet:

```
printf("%s Fehler", string);
```

9.14 FEHLER-PROBLEM NR.14

```
main()
{

    printf("Programmieren...\n in 'C'\n
            hat Zukunft!\n");
    gemdos(0x1);

}
```

Ist dieses Programm nun fehlerhaft oder nicht?

Probieren Sie es doch einmal einfach aus!

Obwohl es reichlich "anormal" aussieht, funktioniert es anstandslos. Es gibt die drei Zeilen

```
Programmieren...
in 'C'
hat Zukunft!
```

untereinander auf dem Bildschirm aus.

Der für den Zeilenschub zuständige New-line-Marker " n" muß nicht wie gewohnt am Ende einer "printf"-Anweisung stehen, sondern kann auch wie in diesem Beispiel in den auszugebenden Text eingefügt werden.

9.15 FEHLER-PROBLEM NR.15

```
main()
{

     int x;

     for(x = 10 ; x > 0 ; --x)
         printf("%d\n", x)
     gemdos(0x1);

}
```

Haben Sie den Fehler entdeckt?

Falls nicht, werden Sie sich wahrscheinlich vor den Kopf schlagen, wenn Sie ihn gleich erfahren.

Es handelt sich um einen banalen Flüchtigkeitsfehler, der sicher jedem 'C'-Programmierer einmal unterläuft, ganz besonders jedoch BASIC-Umsteigern.

Kommen wir zur Lösung. Schauen Sie sich einmal die "printf"-Anweisung in der "for"-Schleife genauer an. An dieser fehlt das abschließende Semikolon, das ist alles.

In BASIC wird das Semikolon nicht benötig. Halten Sie sich daher immer vor Augen, daß das Semikolon in 'C' einzelne Befehle und Operationen voneinander trennt. Es entspricht darin dem Doppelpunkt in BASIC, nur daß das Semikolon, von wenigen Ausnahmen wie der "for"-Anweisung abgesehen, auf jeden (!) Befehl folgen muß.

9.16 FEHLER-PROBLEM NR.16

```
main()
{

     int x = 10;

     while(x > 0)
          {
            printf("Die Zahl ist %d"\n, x);
            --x;
          }
     gemdos(0x1);

}
```

Das Programm führt einen Count-Down von "10" bis "1" durch.

Fehlerhaft ist dabei die "printf"-Zeile.

Das New-line-Steuerzeichen "\n" muß wie alle anderen Steuerzeichen innerhalb der Anführungszeichen stehen.

Die korrekte Anweisung lautet folgendermaßen:

```
printf("Die Zahl ist %d"\n, x);
```

An diesem Beispiel wird auch deutlich, wie man Texte zusammen mit Zahlenvariablen in einer einzigen Anweisung ausgeben kann. Dies ist effizienter als zwei getrennte "printf"-Befehle. Behalten Sie diese Struktur daher im Kopf, Sie werden Sie sicher öfter in Ihren Programmen verwenden.

9.17 FEHLER-PROBLEM NR.17

```
main()
{

     int x;
     scanf("%d\n", &x);

     printf("%d %d\n", x, x*x*x);
     gemdos(0x1);

}
```

Hier wurde ein logischer Fehler begangen.

Er liegt in der "scanf"-Eingabeanweisung. Werfen Sie noch einmal einen genauen Blick auf diese Anweisung.

Fällt Ihnen etwas auf? Genau, das Steuerungszeichen "\n" ist in der Anweisung völlig fehl am Platz.

Dieser Fehler kann entstehen, da "scanf" und "printf" fast identisch in ihrer Syntax sind. Und "printf" verwendet ja sehr viele New-line-Marker "\n". Doch in der "scanf"-Anweisung hat dieser Marker nichts zu suchen, da der Zeilenvorschub nach der Werteeingabe automatisch erfolgt.

Die korrekte Zeile muß also folgendermaßen lauten:

```
scanf("%d", &x);
```

Einige Compiler, wie beispielsweise das DIGITAL C, ignorieren das New-line-Steuerzeichen einfach und erzeugen keinen Fehler.

9.18 FEHLER-PROBLEM NR.18

```
main()
{

      int a, b;
      char *z;

           a = b = 15;

           z = "DIGITAL GEM";

      printf("%d %s %d\n", a, b, z);
      gemdos(0x1);

}
```

Bei diesem Programm müssen Sie etwas genauer hinsehen, um den Fehler zu finden. Er liegt wieder in der "printf"-Anweisung.

Vergleichen Sie die Typensteuerungszeichen mit den zugehörigen Variablen, so werden Sie schnell feststellen, daß die Variable "b" vom Typ Integer ist, jedoch durch "%s" als String angesprochen wird. Entsprechendes gilt für die Stringvariable "z", die durch "%d" als Ganzzahl interpretiert wird.

Die korrekte "printf"-Anweisung muß demnach also lauten:

```
printf("%d %d %s\n", a, b, z);
```

Achten Sie immer darauf, die richtigen Typensteuerungszeichen zu verwenden.

9.19 FEHLER-PROBLEM NR.19

```
main()
{

     int x, y;

         scanf("%d", &x);
         y = 14;

            if(x <> y)
              printf("Der x-Wert ist ungleich dem
                      y-Wert!\n");
            else
              printf("x-Wert ist gleich y-Wert\n");
            gemdos(0x1);

}
```

Der hier dargestellte Fehler ist zwar BASIC-typisch, doch sollten Sie jetzt so viel 'C'-Erfahrung besitzen, daß Sie ihn sofort finden.

Er liegt in der "if"-Anweisung "if(x <> y)". Das BASIC-Symbol für "ungleich" lautet zwar "<>", in 'C' wird daraus jedoch das Zeichen "!=".

Die "if"-Anweisung muß korrekt als

```
if(x != y)
```

formuliert werden.

9.20 ANMERKUNGEN

Dies waren die häufigsten und typischen Fehler von BASIC-Umsteigern. Sicher haben Sie einige dieser Fehler auf Anhieb gefunden, vorausgesetzt Sie haben die vorigen Kapitel gründlich durchgelesen und auch die Beispielprogramme selbst ausprobiert.

Andere Fehler sind Ihnen dagegen wahrscheinlich erst nach dem Durchlesen der Erklärung ersichtlich geworden.

Streichen Sie sich diese Fehler-Probleme am besten im Buch am Rand an. Durch die Markierungen können Sie sich Ihre typischen Fehler bei nochmaligem späteren Durchlesen wieder vergegenwärtigen.

Sie erhalten dadurch einen Anhaltspunkt, worauf Sie sich konzentrieren müssen, wenn Ihr Compiler wieder einmal eine Error-Meldung ausgibt.

Ich hoffe, daß die Fehler-Analysen dieses Kapitels Ihnen viele unnötige Frustrationen bei der Programmerstellung ersparen werden.

TEIL 3:
WEITERFÜHRENDE SPRACHELEMENTE

Kapitel 10:

'C'-Funktionen

Wie immer wieder in diesem Buch hervorgehoben wurde, bestehen alle auch nur etwas umfangreicheren 'C'-Programme aus einer Aneinanderreihung von einzelnen Funktionen.

Eine Funktion stellt dabei ein Unterprogramm dar, vergleichbar mit einer durch GOSUB aufgerufenen und mit RETURN beendeten Subroutine in BASIC.

Selbst wenn in einem 'C'-Programm keine Unterprozedur benötigt wird, so ist es allgemein kein guter Programmierstil, wenn man, wie oft aus BASIC gewohnt, alle Befehle in ein einziges Programm (also in diesem Fall in die "main()"-Funktion) 'quetscht'. Statt dessen ist es sinnvoll, wenn das Programm hier aus lauter einzelnen Funktionen zusammensetzt wird, die von der Funktion 'main()' aus aufgerufen werden.

An diesen anfangs ungewohnten Programmierstil sollten Sie sich möglichst schnell gewöhnen. Vergessen Sie Ihre BASIC-Programmstruktur. Ihre 'C'-Programme gewinnen erheblich an Übersichtlichkeit durch einen konsequenten Programmaufbau aus einzelnen individuellen Funktionen.

Auch sind Programmänderungen innerhalb einer kleinen Funktion erheblich unkomplizierter zu vollziehen, als in einem großen Programm.

Einer der Hauptvorteile der Funktionen ist jedoch, daß Sie sog. "Standard-Funktionen" einsetzen können. Einmal erstellt, kann solche Funktion in jedes neue Programm übernommen werden.

Vor allem aus diesem Grund stellen "Funktionen" einen wesentlichen Kernbestandteil von 'C' dar. Sie haben als Programmierer die Möglichkeit, mittels eigener Funktionen sich eine individuelle Programmiersprache zu erstellen.

Ferner sind auch die Bibliotheken der 'C'-Compiler des ATARI ST zum größten Teil nichts anderes als Funktionen, auf die Sie, beliebigen Zugriff haben. Beispielsweise müssen Sie um GEM in 'C' zu verwenden, lediglich die entsprechenden Bibliotheken in Ihr eigenes Programm einbinden. Ihnen steht damit eine gigantische Bibliothek an sinnvollen Funktionen Verfügung.

Wie sind diese Unterroutinen jedoch genau aufgebaut? Und wie werden sie in Programme eingefügt und aufgerufen?

All dies und viele weitere Detailfragen werden wir auf den folgenden Seiten ganau erörtern und mit vielen 'C'- und BASIC-Beispielen vorstellen.

10.1 GRUNDLAGEN VON FUNKTIONEN

Alle 'C'-Programme können beliebig viele Funktionen enthalten. Mindestens notwendig ist jedoch eine Funktion. Dies ist die Funktion

```
main()
```

die in jedem Programm unbedingt enthalten sein muß. Sie stellt den Programmkopf dar, d.h. sie wird immer als erste in einem Programm aufgerufen.

Generell hat sich eingebürgert, von "main()" aus alle anderen Funktionen eines Programms aufzurufen, d.h. ein Programm befindet sich eigentlich nicht in dieser Hauptfunktion, sondern die Operationen finden in den einzelnen Subroutinen statt.

Die Funktion 'main()' dient dabei lediglich zum Verwalten und Steuern dieser Funktionen.

10.1.1 AUFRUFEN VON FUNKTIONEN

Schauen Sie sich einmal das folgende Beispiel an. In diesem ruft die Funktion "main()" sich immer wieder selbst auf:

```
main()
{

      printf("Hallo, wie gehts?\n");

      main();

}
```

Dieses Programm gibt die folgende Meldung aus:

```
Hallo, wie gehts?
Hallo, wie gehts?
Hallo, wie gehts?
Hallo, wie gehts?
Hallo, wie gehts?
...
```

Der Text wird in einem sich endlos wiederholenden Aufruf immer wieder auf dem Bildschirm ausgegeben.

Um das Programm zu beenden, müssen Sie CTRL-C drücken und Sie befinden sich wieder in 'C'.

Halten wir vorerst fest: In diesem Beispiel ruft die Funktion "main()" sich einfach selbst durch den Programmbefehl

```
main();
```

auf. Ein Aufruf einer Funktion geschieht demnach allgemein durch ihren Funktions-Namen im Programm. In den hier noch leeren Argumentklammern der Funktion können Parameter an die Funktion übergeben werden.

10.1.2 BEISPIELE EINER FUNKTION OHNE PARAMETER-ÜBERGABE

Wenden wir die bisher erwähnten Fakten einmal in einem Demoprogramm an.

Hier kommt ein BASIC-Beispiel:

```
10 PRINT "BASIC"
20 GOSUB 1000
30 :
40 PRINT "FORTH"
50 GOSUB 1000
60 :
70 PRINT "LISP"
80 GOSUB 1000
90 :
100 PRINT "PROLOG"
110 GOSUB 1000
120 :
130 PRINT "'C'"
140 GOSUB 1000
150 :
160 :
1000 REM UNTERPROGRAMM "KEYSTOP"
1005 :
1010 PRINT "DRÜCKEN SIE EINE TASTE..."
1020 GET A$
1030 IF A$="" THEN 1020
1035 :
1040 RETURN
```

In diesem Programm werden einige Programmiersprachen hintereinander ausgegeben. Nach jeder Sprache springt das Programm über GOSUB in die Subroutine "KEYSTOP".

In der wird die Meldung

```
"DRÜCKEN SIE EINE TASTE..."
```

ausgegeben und auf einen Tastendruck gewartet. Ist dieser erfolgt, so wird ein RETURN in das Hauptprogramm vorgenommen.

Vergleichen Sie das BASIC-Programm einmal mit der 'C'-Version:

```
#include "stdio.h"
#define getchar() getc(stdin)
main()
{

    printf("BASIC\n");
    keystop();

    printf("FORTH\n");
    keystop();

    printf("LISP\n");
    keystop();

    printf("PROLOG\n");
    keystop();

    printf("'C'\n");
    keystop();

    gemdos(0x1);

}

keystop()
{
      int a;
```

```
        printf("Drücken Sie eine Taste...\n");
        getch(a);
}

char bf[100];
int b = 0;
getch()
{
     return((b > 0) ? bf[--b] : getchar());
}
```

Auf der vorherigen Seite sehen Sie die aus dem Hauptprogramm "main()" über den Befehl

```
keystop();
```

aufgerufene Funktion "keystop()". Sie definiert zunächst eine "lokale Variable" vom Typ Integer.

Anschließend erfolgt die Ausgabe

```
"Drücken Sie eine Taste..."
```

und das Programm wartet durch den Befehl 'getch(a);' auf die Eingabe eines beliebigen Zeichens über die Tastatur. Danach erfolgt automatisch der Sprung zurück in das Hauptprogramm. Auf dem DIGITAL 'C' ist das Betätigen von RETURN notwendig.

Sie sehen, zwischen BASIC-und 'C'-Unterprogrammen gibt es fast keine Unterschiede. Der Aufruf einer Routine in 'C' erfolgt lediglich über den Funktionsnamen, in BASIC dagegen wird meistens die Zeilennummer des ersten Befehls des Unterprogramms benötigt.

Die Anweisung RETURN zum Beenden der Funktion kann in 'C' entfallen. Der Rücksprung in die Funktion, von der aus die Unterfunktion aufgerufen wurde, in diesem Fall der Funktion 'main()', erfolgt automatisch nach dem letzten Befehl, also in der Beispielfunktion nach der Anweisung 'getch(a);'.

Wir hätten die Funktion "keystop" auch beliebig anders formulieren können. Möglich wäre beispielsweise auch eine PAUSE-Funktion oder WAIT-Anweisung, wie sie in einigen BASIC-Versionen implementiert ist.

Das Unterprogramm unseres BASIC-Beispiels lautet dann

```
1000 REM PAUSE-UNTERPROGRAMM "KEYSTOP"
1005 :
1010 PAUSE 1000
1020 RETURN
```

oder, wenn Ihr BASIC den Befehl "PAUSE n" nicht besitzt, folgendermaßen:

```
1000 REM PAUSE-UNTERPROGRAMM "KEY-STOP"
1005 :
1010 FOR A = 1 TO 10000: NEXT A
1020 RETURN
```

Die beiden Unterprogramme bewirken eine gewisse Zeitverzögerung, bevor der Name der nächsten Programmiersprache im Hauptprogramm ausgegeben wird.

In 'C' sähe das entsprechende Funktions-Unterprogramm dann so aus:

```
keystop()
{
      int x;

      for(x = 1; x < 30000; ++x)
           ;
}
```

Diese PAUSE-Routine läßt sich übrigens genau wie die vorherige TASTEN-Funktion gut in Ihren eigenen Programmen verwenden.

10.1.3 FUNKTIONEN RUFEN SICH GEGENSEITIG AUF

Sehen Sie sich einmal das folgende 'C'-Programm an:

```
main()
    {
        M();
        gemdos(0x1);
    }

e()
    {
        putchar('e');
        s();
        putchar('e');
    }

M()
    {
        putchar('M');
        e();
    }

s()
    {
        printf("ss");
    }
```

An diesem Beispiel wird deutlich, wie sich einzelne Funktionen in 'C' auch gegenseitig aufrufen können.

Auch in BASIC ist das genauso ohne weiteres möglich. Das obige 'C'-Programm sieht dann etwa folgendermaßen aus:

```
10 GOSUB 100
20 END

100 PRINT "M";
110 GOSUB 200
120 RETURN

200 PRINT "E";
210 GOSUB 300
220 PRINT "E";
230 RETURN

300 PRINT "SS";
310 RETURN
```

Haben Sie herausgefunden, was diese Programme bewirken? Genau, durch die gegenseitigen Unterprogramm-Aufrufe wird das Wort

```
Messe
```

bzw.

```
MESSE
```

auf dem Bildschirm ausgegeben. An diesem Beispiel wird hervorragend ersichtlich, wie sich Funktionen in 'C' gegenseitig und auch verschachtelt aufrufen können.

10.2 PARAMETERÜBERGABE AN FUNKTIONEN

Bisher scheinen sich die Funktionen von 'C' nur geringfügig von den Subroutinen in BASIC zu unterscheiden. Dies gilt jedoch nur für den Fall, daß keine Parameter an die Funktion übergeben werden.

Dieser Fall wurde in den Beispielen auf den vorigen Seiten dargestellt. In der von "main()" aus aufgerufenen Funktion "keystop" wurde nur ein fester PAUSE-Zeitverzug bewirkt.

Angenommen Sie wollen jedoch z.B. einen

```
PAUSE n
```

Befehl, wie er in einigen BASIC-Versionen vorhanden ist, verwirklichen. Die Variable 'n' stellt dabei die Pausenlänge dar.

Bei der Verwirklichung in 'C' muß dabei der Parameter 'n' an die Funktion weitergegeben werden.

Die Funktion sieht dann folgendermaßen aus:

```
pause(n)
int n;
{

      int a;
      for(a = 1; a < n; ++a)
            ;

}
```

Der Funktion 'pause()' wird durch die Variable (n) innerhalb der Klammern angezeigt, daß sie den Parameter (n) übernehmen soll.

Der Funktionsaufruf erfolgt durch die Anweisung:

```
pause(10000);
```

oder

```
pause(40000);
```

Je nach dem Wert des übergebenen Parameters ist die "for"-Schleife mehr oder weniger groß, die Pause also kürzer oder länger, genau entsprechend dem BASIC-Befehl.

Schauen wir uns die Anwendung unserer neuen "pause(n)"-Funktion einmal genauer in einem Beispielprogramm an:

```
main()
{

     printf("Hallo, ");
     pause(30000);

     printf("wie...");
     pause(20000);

     printf(" gehts?");
     gemdos(0x1);

}

pause(n)
int n;
{
      int a;
      for( a = 1; a < n; ++a)
           ;
}
```

Um Ihnen eine Vergleichsmöglichkeit zu geben, kommt jetzt das entsprechende Programm in BASIC.

Dabei wurde davon ausgegangen, daß die verwendete BASIC-Version den **PAUSE_N** -Befehl nicht besitzt, dieser also nachgebildet werden muß:

```
10 PRINT "HALLO, ";
20 N=3000: GOSUB 1000

30 PRINT "WIE...";
40 N=2000: GOSUB 1000

50 PRINT " GEHTS?"
60 END

1000 REM PAUSE N
1010 FOR A = 1 TO N: NEXT A
1020 RETURN
```

In BASIC sind direkte Parameterübergaben an Unterprogramme nicht möglich. Sie müssen indirekt, wie in diesem Beispiel, durch globale Variablenwerte weitergegeben werden.

In diesem Fall stellt die Variable "N" die Pausenlänge dar, die die FOR-NEXT-Schleife in dem Unterprogramm ab Zeile 1000ff steuert.

10.2.1 RETURN VON INTEGER-DATEN

Bisher haben wir uns mit der Form, dem gegenseitigen Aufrufen sowie der Parameterübergabe von Funktionen befaßt.

Wollen Sie jedoch erreichen, daß eine Funktion Variablenwerte an die sie aufrufende Funktion zurückgibt, so müssen Sie, verglichen mit BASIC, einen etwas ungewohnten Weg beschreiten:

Nehmen wir als Beispiel das folgende BASIC-Programm, das den Kubikwert einer Variable ermittelt:

```
10 INPUT X%
20 GOSUB 100
30 PRINT Y%
40 END
50 :
100 Y% = X% * X% * X%
110 RETURN
```

Kommen wir nun zu der entsprechenden 'C'-Realisierung mittels zweier Funktionen:

```
main()
{

     int x,
         y;

     scanf("%d", &x);
     y = kubik(x);
         printf("Der Kubikwert ist %d\n", y);

     gemdos(0x1);

}

kubik(z)
int z;
{

    return(z * z * z);

}
```

In 'C' muß der Datenwert von x^3 mittels der "return"-Anweisung an "main()" übergeben werden.

Dabei bewirkt die Anweisung

```
y = kubik(x)
```

daß der Variablen "y" genau der Wert zugewiesen wird, der durch die Anweisung "return" in der Funktion "kubik()" an "main()" zurückgegeben wurde.

Dies ist besonders für BASIC-Umsteiger sehr gewöhnungsbedürftig, bietet dennoch Vorteile, die im Laufe des Kapitels noch deutlich werden.

Ich möchte Ihnen zum anschaulichen Vergleich jetzt zeigen, wie Sie Funktionen **nicht** anwenden dürfen:

```
main()
{

     int x,y;

     scanf("%d", &x);
     y = quadrat(x);

     printf("%d\n", y);
     gemdos(0x1);

}

quadrat(q)
int q;
{

      q = q * q;

}
```

Das Programm stellt einen ganz typischen Fehler von BASIC-Programmierern dar. In BASIC würde ein derartiges Unterprogramm fehlerfrei funktionieren, in 'C' dagegen nicht. Warum?

Es verändert lediglich den Wert einer Variablen, weist diesen Wert jedoch nicht an eine Speicheradresse zu. Um das zu erreichen, könnten Sie beispielsweise die auf der vorigen Seite vorgestellte "return"-Anweisung in das Programm einfügen.

Wir haben in diesem Buch allerdings auch schon eine andere Möglichkeit kennengelernt, um Variablenwerte einer speziellen Speicheradresse zuzuweisen. Dies geschieht über Pointervariablen.

Wie Sie Zeiger einsetzen können, um Variablen beliebig zwischen einzelnen Funktionen auch ohne die Anweisung 'return' auszutauschen und was Sie sonst noch mit Zeigern in Funktionen bewirken können, wird einige Seiten weiter noch genauer beschrieben.

10.2.2 RETURN ANDERER NUMERISCHER DATENTYPEN

Wenn wir mittels "return" andere Variablentypen als Integer-Werte an Funktionen übergeben wollen, so muß eine Besonderheit beachtet werden.

Wandeln wir zur näheren Erklärung unser voriges BASIC-Programm so ab, daß der Quadratwert einer eingegebenen Zahl nicht nur von einer Integer, sondern auch von einer 'float'-Variable ermittelt werden kann:

```
10 INPUT X
20 GOSUB 100
30 PRINT Y
40 END
50 :
100 Y=X*X*X
110 RETURN
```

Die 'C'-Realisation muß dann folgendermaßen lauten:

```
main()
{

     float x,
           y,
           kubik();

             scanf("%f", &x);

             y = kubik(x);
         printf("%f\n", y);
     gemdos(0x1);

}


float kubik(z)
float z;
{

      return(z * z * z);

}
```

Sie sehen an dem Programm, daß als erstes der 'float'-Variablentyp zu Beginn der Funktion 'main()' deklariert werden muß, wenn dieser Wert von der Funktion 'kubik()' an die Funktion 'main()' zurückgegeben werden soll.

Dies geschieht über die Deklarationsanweisung:

```
float kubik();
```

Dieser Variablentyp muß dann auch in der Funktion 'kubik()' erneut als 'float'-Wert deklariert werden, wie dies auch schon bei Integer-Zahlen erforderlich war.

Sicher ist Ihnen auch aufgefallen, daß die Sub-Funktion nicht mehr wie gewohnt mit dem einfachen Namen

```
kubik()
```

sondern mit dem Funktionskopf

```
float kubik()
```

eröffnet wird. Sie müssen also den Variablentyp erneut vor dem Funktionsnamen anführen.

In der Praxis werden Sie jedoch nur wenige "float"-Funktionen in 'C'-Programmen finden oder selber einführen. Statt dessen werden die meisten Funktionen, die keine "Integer"-Werte zurückgeben, als "double"-Funktionen deklariert, um die erhöhte Rechengenauigkeit der "double"-Variablen zu nutzen.

10.2.3 ZEIGER UND FUNKTIONEN SOWIE SIMULTANE PARAMETERÜBERGABE

Pointer spielen beim Datentransfer zwischen den Funktionen eine sehr wichtige Rolle. Schauen wir uns den Einsatz von Zeigern an einem BASIC-Programm einmal näher an:

```
10 INPUT A
20 INPUT B
30 GOSUB 1000
40 PRINT A;B
50 END
100 :
1000 REM TAUSCH
1010 HI=A
1020 A=B
1030 B=HI
1040 RETURN
```

Hier wird in der Subroutine "TAUSCH" ein Variablentausch unter den Variablen "A" und "B" durchgeführt. In einigen BASIC-Versionen ist dafür ein spezieller "SWAP"-Befehl vorhanden. Dieser entspricht genau der verwendeten Unterroutine. Ein derartiger Befehl ist sehr nützlich und läßt sich z.B. in Sortierprogrammen einsetzen.

Kommen wir nun zu der Realisierung in 'C' mittels Pointer:

```
main()
{

     int a,b;
        scanf("%d %d", &a, &b);
        tausch(&a, &b);
        printf("%d %d\n", a, b);
     gemdos(0x1);

}
tausch(c, d)
int *c, *d;
{

      int hilfsvariable;

         hilfsvariable = *c;
         *c = *d;
         *d = hilfsvariable;

}
```

Beim Vergleich der beiden Unterprogramme in BASIC als auch in 'C' werden Sie schnell feststellen, wieviel Ähnlichkeit zwischen den jeweiligen Realisierungen besteht. Sie können anhand der Zeiger fast genauso wie aus BASIC gewohnt 'C'-Unterprogramme formulieren.

Lediglich die Festlegung der einzelnen Pointer ist zu beachten. Natürlich müssen auch, wie in 'C' allgemein üblich, alle lokalen Variablen innerhalb einer Funktion jeweils deklariert werden.

Pointer wurden in diesem Buch bereits ausführlich erklärt. Um mich nicht zu wiederholen, fasse ich hier nur noch kurz die spezielle Anwendung von Zeigern im Einsatz zwischen Funktionen am Beispiel dieses Programms zusammen:

Beachten Sie zunächst den **Aufruf der Tauschfunktion** über den Befehl:

```
tausch(&a, &b);
```

Die Variablen "a" und "b" müssen unbedingt mit dem Adreßoperator "&" versehen werden, da deren Werte ja von der Funktion "tausch()" aus über Speicheradressen verändert werden sollen. Mittels "&a" und "&b" werden die Speicheradressen beider Variablen an die Funktion "tausch()" übergeben.

Innerhalb dieser Funktion müssen die Variablen "c" und "d" als Integer-Pointer deklariert werden. Damit erhalten Sie den Inhalt der Variablen "a" und "b" aus "main()" zugewiesen. Die beiden Pointer werden anschließend ausgetauscht. Die dabei verwendete "hilfsvariable" muß kein Pointer sein.

Beide so geänderten Variablen werden als Integer-Pointer zurück an die Funktion 'main()' gegeben. Dadurch erfolgte ein Austausch der Inhalte der Variablen "a" und "b".

In diesem Beispiel wurden erstmals zwei Parameter gleichzeitig an eine Funktion zugewiesen, als auch wieder zurückgegeben. Mittels "return" ist lediglich die Rückgabe eines einzelnen Wertes möglich.

Nur über den Einsatz von Zeigern läßt sich wie in diesem Beispiel die Rückgabe mehrerer Werte ermöglichen.

Da die Vorabversion des DIGITAL 'C' des ATARI-ST-Entwicklungssystems wie bereits erörtert Pointeroperationen nicht korrekt ausführt, funktioniert dieses Unterprogramm nur zum Teil.

Die nicht standardgemäßen Zeigeroperationen sind der größte Mangel dieses Compilers, der vor seiner kommerziellen Veröffentlichung unbedingt behoben werden muß. Auf allen anderen Compilern funktionieren die Zeigeroperationen wie sie hier beschrieben wurden.

10.3 DER BEFEHL DEF FN

In BASIC ist es möglich, über "DEF FN"-Funktionen zu definieren. Das folgende Beispielprogramm führt eine Kubik-Funktion ein und wendet sie anschließend an:

```
10 DEF FNKUBIK(X) = X*X*X
20 :
30 INPUT X:
40 Y=FNKUBIK(X): PRINT Y
50 END
```

In 'C' können wir eine Funktion über die bereits im Zusammenhang mit Konstanten vorgestellte "#define"-Konstruktion definieren. Unser BASIC-Programm wird dann in 'C' zu

```
#define kubik(x)  x * x * x
main()
{

        int x, y;

           scanf("%d", x);
           y = kubik(x);
           printf("%d\n", y);

        gemdos(0x1);

}
```

Hier stellt die Anweisung

```
#define kubik(x) x * x * x
```

ein sog. Makro dar. **Makros** sind schnell und einfach geschrieben. Ihr großer Vorteil liegt in Ihrer unkomplizierten Struktur.

Vergleichen Sie den "kubik"-Makro einmal mit der entsprechenden herkömmlichen Funktion:

```
kubik(x)
int x;
{

      return( x * x * x);

}
```

Sie sehen, daß diese Funktion umfangreicher und aufwendiger als der Makro ist. Aus diesem Grund sind Makros auch äußerst häufig in 'C'-Programmen anzutreffen.

Wie funktioniert ein Makro? Die **Funktionsweise** ist völlig identisch mit der der bereits früher im Buch beschriebenen symbolischen Konstanten.

Dort wurde festgestellt, daß der Compiler bei der Übersetzung eines Programms den Inhalt der "#define"-Ausdrücke einfach an den entsprechenden Stellen in die Funktionen einsetzt.

Aus diesem Grund können nicht nur Funktionen, sondern auch Befehle als Makros abgefaßt werden. Schauen wir uns das einmal an einem Beispiel an:

```
#define printfs(x)  printf("%s\n", x);
main()
{
```

```
        char *a;
        a = "Eingabe...>";

        printsf(a);

}
```

Dieses Programm setzt die Makros in einer völlig neuen Form ein, wie sie beispielsweise in BASIC nicht möglich ist.

Bei der Übersetzung des Programms geht der Compiler nun so vor,daß er sobald er an den Befehl

```
printsf(a);
```

kommt, diesen durch die vorher definierte Anweisung

```
printf("%s\n", a);
```

ersetzt.

Ein derartiger Einsatz von Makros ist äußerst leistungsfähig und stellt bei der Programmerstellung eine große Vereinfachung für Sie dar.

Schreiben Sie doch einmal zur Übung Ihre eigenen Makros.

Kapitel 11:

Verbundstrukturen

Sturukturen sind in BASIC nicht verfügbar, jedoch in vielen anderen Programmiersprachen. In Pascal beispielsweise nennt man diesen Variablentyp **"Record"**. Der Aufbau von Strukturen ist aber nicht allzu schwer zu verstehen.

Auf den folgenden Seiten beschreibe ich alles wesentliche, was Sie als BASIC-Umsteiger über Strukturen, deren Syntax sowie ihrer Verwendung in Programmen wissen müssen.

Ich möchte jedoch schon gleich am Anfang dieses Kapitels etwas beruhigen. Meiner Ansicht nach sind Strukturen speziell für BASIC-Programmierer auf jeden Fall anfangs nicht notwendig. Sie können jedes beliebige 'C'-Programm, wie bisher in dem Buch beschrieben, mit den herkömmlichen Variablentypen formulieren.

Mit zunehmender Praxis-Erfahrung werden Sie dann feststellen, daß sich umfangreichere Algorithmen mittels Strukturen **vereinfachen** und **übersichtlicher** darstellen lassen.

Doch wenden wir uns nun direkt den Strukturen zu.

11.1 DEKLARATION VON STRUKTUREN

Was sind denn nun Strukturen? Sie stellen einen **Verbundvariablentyp** dar. Das bedeutet konkret, daß unter einem Strukturnamen verschiedene Variablen, auch mit unterschiedlichen Variablentypen, definiert werden.

Die Deklaration erfolgt in einer eigenen Routine. Schauen wir uns dies einmal an einem Beispiel an:

```
struct artikel
       {
           int nummer;
           char *bezeichnung
           float preis
       };
```

In der Struktur mit dem Titel bzw. dem 'Kenner' 'artikel' wird ein Geschäftsartikel erfaßt. Die einzelnen Elemente der Struktur sind die 'artikel'-"nummer", die 'artikel'-"bezeichnung" sowie der 'artikel'-"preis".

An der hier verwendeten Schreibweise wird deutlich, was sich unter der vorher erwähnten Bezeichnung von Strukturen als "Verbundvariablen" verbirgt: Der Strukturtitel "artikel" stellt einen Ober- bzw. Sammelbegriff für die einzelnen Artikelkennzeichen dar.

Eine derartige Strukturdeklaration steht normalerweise ganz am Anfang eines Programms, noch bevor die globalen oder lokalen Variablen definiert werden.

Achten Sie hier auch auf das Semikolon nach der Deklaration, welches immer auf die schließende Klammer folgen muß.

11.2 ANWENDUNG VON STRUKTUR-VARIABLEN

Bleiben wir bei unserem Artikelstruktur-Beispiel. Durch die eben vorgenommene Deklaration wurde lediglich die Form der Struktur 'artikel' festgelegt.

Anders ausgedrückt, Sie haben bestimmt, daß das erste Element der Struktur die "nummer", das zweite die "bezeichnung" und das dritte den "preis" darstellt. In einer Tabelle sieht dies so aus:

```
      FORM: nummer(int)  bezeichnung(*char)  preis(float)
-------------------------------------------------------
Artikel Nr.1      ...        ...               ...
Artikel Nr.2      ...        ...               ...
Artikel Nr.3      ...        ...               ...
   ...
Artikel Nr.n      ...        ...               ...
```

Die Kopfzeile stellt hier die Form, der Struktur "artikel" dar, festgelegt durch ihre vorherige Strukturdeklaration.

Wie am linken Rand dargestellt ist, kann diese Form mit beliebig vielen Artikeln aufgefüllt werden.

Wieviele Artikel Sie wünschen und wie diese genau als Variablen angesprochen werden sollen, all dies wird in der Strukturvariablendeklaration festgelegt.

Sollen beispielsweise vorerst nur drei Artikel (nr_1 bis nr_3) als Variablen in der Struktur ablegt werden, so ist dazu die folgende Anweisung nötig:

```
struct artikel nr_1, nr_2, nr_3;
```

Die Tabelle stellt sich dann nach dieser Vereinbarung folgendermaßen dar:

```
   nummer(int)      bezeichnung(*char)   preis(float)
-------------------------------------------------------
   nr_1.nummer      nr_1.bezeichnung     nr_1.preis
   nr_2.nummer      nr_2.bezeichnung     nr_2.preis
   nr_3.nummer      nr_3.bezeichnung     nr_3.preis
```

Hier können Sie nun sehen, wie die einzelnen Elemente der Struktur "artikel" angesprochen werden. Die allgemeine Formulierung ist:

```
Strukturvariablenname.Elementname
```

Zunächst müssen Sie also den Strukturvariablennamen, in diesem Fall "nr_1", "nr_2" oder "nr_3" angeben.

Dann folgt ein Trennungspunkt.

Anschließend müssen Sie eines der in der Strukturdeklaration festgelegten Elemente anführen. In diesem Beispiel wurden als Form die Variablen "name", "bezeichnung" sowie "preis" eingeführt.

Die so festgelegten Strukturvariablen lassen sich wie normale Variablen einsetzen. Möglich ist also beispielsweise die Anweisung

```
nr_2.nummer = 125;
```

oder auch

```
nr_1.bezeichnung = "ATARI 520ST";
```

Wie die Struktur-Elemente innerhalb eines Programms eingesetzen werden, veranschaulicht das folgende kurze Beispielprogramm:

```
struct artikel
{

     int nummer;
     char *bezeichnung;
     float preis;

};

main()
```

```
{

    struct artikel nr_1,nr_2,nr_3;

    nr_1.nummer = 125;
    nr_1.bezeichnung = "ATARI 520ST";
    nr_1.preis = 2750.0;

    nr_2.nummer = 15;
    nr_2.bezeichnung = "Commodore C-128";
    nr_2.preis = 998.0;

    nr_3.nummer = 548;
    nr_3.bezeichnung = "Amiga";
    nr_3.preis = 4000.0;


       printf("Der Artikel Nr.1 ist ... %d\n",
              nr_1.nummer);
       printf("Der Artikel Nr.2 ist ... %s\n",
              nr_2.bezeichnung);
       printf("Der Preis Nr.1 beträgt ... %f\n",
              nr_1.preis);

    gemdos(0x1);

}
```

Anhand der "main()"-Funktion auf dieser Seite wird zweierlei dargestellt:

Zum einen wird noch einmal klar deutlich, wie die Strukturvariablendeklaration vorgenommen wird, zum anderen ist die genaue Anwendung dieser Variablen im Programm beispielhaft veranschaulicht.

11.3 DATENFELDER UND STRUKTUREN

Sie können jetzt bereits Strukturen und die zugehörigen Strukturvariablen problemlos in Ihren Programmen einsetzen.

In diesem Abschnitt lernen Sie nun eine Vereinfachung im Umgang mit den Strukturvariablen kennen. In dem vorigen Beispiel wurden die Artikelnummern, also die Strukturvariablennamen mit "nr_1", "nr_2" und "nr_3" benannt.

In der Praxis wird man eine derartige Formulierung wohl kaum finden oder anwenden. Die Artikelnummern lassen sich kürzer und effektiver über Arrays ansprechen.

Die Strukturvariablendeklaration in der "main()"-Funktion läßt sich mittels eines Feldes so gestalten

```
struct artikel nr[3];
```

anstatt wie bisher mittels

```
struct artikel nr_1,nr_2,nr_3;
```

Vor allem bei den in den meisten Fällen angewendeten erheblich umfangreicheren Strukturen stellt die Arrayformulierung eine erhebliche Vereinfachung dar.

Unsere Demo-Tabelle sieht nach der neuen Variablenstrukturdeklaration folgendermaßen aus:

```
  name(int)     bezeichnung(*char)     preis(float)
-----------------------------------------------------
 nr[0].name     nr[0].bezeichnung      nr[0].preis
 nr[1].name     nr[1].bezeichnung      nr[1].preis
 nr[2].name     nr[2].bezeichnung      nr[2].preis
```

Erweitern wir nun einmal unser voriges 'C'-Programm mit einem Datenfeld:

```
struct artikel;
{
    int nummer;
    char *bezeichnung;
    float preis;
};

main()
{

      int n;
      struct artikel nr[3];

      nr[0].nummer = 125;
      nr[0].bezeichnung = "ATARI 520ST";
      nr[0].preis = 2750.0;

      nr[1].nummer = 15;
      nr[1].bezeichnung = "Commodore C-128";
      nr[1].preis = 1200.0;

      nr[2].nummer = 548;
      nr[2].bezeichnung = "Amiga";
      nr[2].preis = 4000.0;

      printf("NummerBezeichnungPreis \n");

      for(n = 0; n < 3; ++n)
         {
            printf("%d   %s   %fn",
            nr[n].nummer, nr[n].bezeichnung,
            nr[n].preis );
         }
      gemdos(0x1);

}
```

An diesem Beispiel wird deutlich, warum nahezu alle Stukturen als Arrays definiert werden:

Die 'for'-Schleife, welche alle Elemente der Struktur ausgibt, läßt sich nur mittels der Zuhilfenahme von Feldern einsetzen. Nur so sind die allgemeinen Aufrufe

```
nr[n].nummer
```

```
nr[n].bezeichnung
```

und

```
nr[n].preis
```

möglich.

Auch nachträgliche Änderungen sind ohne Probleme realisierbar. Soll beispielsweise die Struktur "artikel" von derzeit drei auf beispielsweise 100 Artikel-Elemente erweitert werden, so muß lediglich die Variablenstruktur-Deklaration geändert werden.

Aus

```
struct artikel nr[3];
```

wird dann die Deklaration

```
struct artikel nr[100];
```

Nachdem Sie die hier vorgestellten Demo-Programme ausprobiert haben, sollten Sie am besten mit den Strukturen in eigenen Programmen experimentieren, um so Ihr Wissen zu festigen.

TEIL 4:
NACHSCHLAGETEIL

Kapitel 12:

Ein 'C'-Überblick

12.1 KEYWÖRTER IN 'C'

Auf den folgenden Seiten erhalten Sie eine prägnante Zusammenfassung der wichtigsten 'C'-Sprachelemente.

Der Grundwortschatz von 'C' lautet folgendermaßen:

SCHLEIFENANWEISUNGEN:

```
for
do
while
```

ENTSCHEIDUNGSANWEISUNGEN:

```
if
else
switch
case
default
```

SPRUNGANWEISUNGEN:

```
break
continue
goto
```

SPEICHERKLASSEN:

```
auto
extern
static
```

DATENTYPEN:

```
int
short
long
unsignet
float
double
char
struct
```

WEITERE ANWEISUNGEN:

```
return
exit
type
define
include
printf
scanf
```

Diese für Umsteiger wichtigsten Schlüsselwörter stellen den Sprachkern von 'C' dar. Hier läßt sich deutlich erkennen, daß der wesentliche Sprachumfang noch geringer als der von BASIC ist.

Die eigentlichen Probleme für 'C'-Newcomer treten aber bei der vielfältigen Anwendungsmöglichkeit jeder einzelnen Funktion auf.

Wenn man zu diesem festen Kern noch die verwirrend große Anzahl an zusätzlichen Funktionen der Bibliotheken dazunimmt, kommt man schnell auf einen minimalen Sprachumfang von 200 Befehlen, Funktionen, Steuerzeichen und Operatoren für einen normalen 'C'-Compiler auf dem PC. Der für Sie wichtige Sprachkern, aus dem sich die Zusatzfunktionen zusammensetzen, ist im folgenden dargestellt:

12.2 DIE 'C'-SPRACHANWEISUNGEN:

In dieser Zusammenstellung erhalten Sie auf einen Blick die wichtigsten Sprachanweisungen mit deren Syntax und wichtigen Besonderheiten bei ihrer Implementierung in 'C'-Programmen:

12.2.1 DIE 'BREAK'-ANWEISUNG

Syntax:

```
break;
```

Die 'break'-Anweisung wird immer dann eingesetzt, wenn die Ausführung einer 'do'-, 'for'-, 'switch'- oder 'while'-Anweisung sofort beendet werden soll. Nach dem 'break'-Sprung wird das Programm entweder mit der folgenden Schleife oder aber, falls vorhanden, der 'switch'-Anweisung weitergeführt.

12.2.2 DIE 'CASE'-ANWEISUNG

Syntax:

```
case Konstante:
        Anweisung 1
        Anweisung 2
        ...
```

Die 'case'-Anweisung ist Bestandteil der 'switch'-Verzweigungsstruktur. Je nach dem x-Wert der 'switch(x)'-Anweisung wird der 'case'-Block ausgeführt, dessen Konstantenwert dem x-Wert entspricht.

12.2.3 DIE 'CONTINUE'-ANWEISUNG

Syntax:

```
continue;
```

Die 'continue'-Anweisung wird innerhalb einer Schleife eingesetzt. Dabei werden alle Anweisungen, die nach 'continue' in dieser Schleife folgen, nicht mehr ausgeführt. Statt dessen wird direkt mit der Ausführung des auf die verlassene Schleife folgenden Befehls begonnen.

12.2.4 DIE '#DEFINE'-ANWEISUNG

Syntax 1:

```
#define Name Ersatztext
```

Bei Verwendung des über die #define-Konstruktion festgelegten Namens in einem 'C'-Programm wird dieser bei der Compilierung durch den Ersatztext ersetzt.

Syntax 2:

```
#define Name(Parameter 1, Parameter 2,
             Parameter n) Text
```

Das Makro 'Name' wird definiert und übernimmt die 'n'-Parameter in die Anweisung 'Text', welcher an die Stelle von 'Name' in das Programm übernommen wird. Ein Beispiel dafür ist das Makro

```
#define quadrat(x) x * x
```

Der Aufruf des Makros durch beispielsweise

```
x = quadrat(wert_1 * 2);
```

bewirkt das Ersetzen des Makros im Programm durch die Anweisung

```
x = (wert_1 * 2) * (wert_1 * 2);
```

12.2.5 DIE 'DEFAULT'-ANWEISUNG

Syntax:

```
default:
      Programm-Anweisung
```

Die 'default'-Anweisung bewirkt innerhalb einer 'switch'-Anweisung, daß, wenn keine 'case'-Bedingung erfüllt war, die auf 'default' folgenden Programm-Anweisungen ausgeführt werden.

Beinhaltet eine 'switch'-Anweisung keinen 'default'-Teil und wurde keine 'case'-Bedingung erfüllt, so wird die 'switch'-Struktur ohne eine Ausführung der in 'switch' enthaltenen Anweisungen verlassen.

12.2.6 DIE 'DO'-ANWEISUNG

Syntax:

```
do
       {
       Programm-Anweisungen
       }
while (Bedingung);
```

Die Programmanweisungen innerhalb der 'do'-Schleife werden solange ausgeführt, bis die Bedingung der 'while'-Anweisung erfüllt wird.

Hierbei ist zu beachten, daß die nach 'do' folgende Anweisung auf jeden Fall einmal ausgeführt wird, da die Bedingungsabfrage durch 'while' erst am Ende der Schleife erfolgt.

12.2.7 DIE 'ELSE'-ANWEISUNG

Syntax:

```
else
   {
      Programm-Anweisungen
   }
```

Die 'else'-Anweisung ist ein Bestandteil der 'if'-Verzweigung. Der 'else'-Befehlsblock wird ausgeführt, wenn die Bedingung der 'if'-Anweisung unwahr ist, also dem Wert Null entspricht.

12.2.8 DIE 'ELSE IF'-ANWEISUNG

Syntax:

```
else if (Bedingung)
     {
        Programm-Anweisungen
     }
```

Die 'else if'-Anweisung folgt auf eine 'if'- oder eine vorangegangene 'else if'-Anweisung. Ist die 'else if'-Bedingung wahr, d.h. der Wert der Bedingung ungleich null, so werden die Programm-Anweisungen ausgeführt.

12.2.9 DIE 'FOR'-ANWEISUNG

Syntax:

```
for (erste Intervallgrenze, zweite Grenze, Schrittweite)
    {
    Programm-Anweisungen
    }
```

Der Parameter **'erste Intervallgrenze'** initialisiert eine Schleifenvariable. Der hier festgelegte Wert stellt die linke Grenze der Schleife fest. Der Parameter 'zweite Grenze' legt die Bedingung für die Beendung der 'for'-Schleife fest.

Die Schrittweite beeinhaltet die schrittweise Veränderung der Schleifenvariablen. Die Programm-Anweisungen werden innerhalb der 'for'-Schleife wiederholt.

12.2.10 DIE 'GOTO'-ANWEISUNG

Syntax:

```
goto Bezeichnung;
```

Die Ausführung der 'goto'-Anweisung bewirkt einen direkten Sprung an die Stelle des Programms, die mit

```
Bezeichnung: Anweisungen
```

adressiert ist.

12.2.11 DIE 'IF'-ANWEISUNG

Syntax:

```
if (Bedingung)
   {
      Programm-Anweisungen
   }
```

Wenn die Bedingung wahr ist, d.h. ihr Wert ungleich null ist, so werden die Programm-Anweisungen ausgeführt.

12.2.12 DIE LEERE ANWEISUNG

Syntax:

```
;
```

Diese Anweisung hat keinerlei Auswirkungen auf den Programmablauf. Sie muß aber oft eingesetzt werden, um die geforderte Syntax einer Anweisung zu erfüllen. Anwendungen ergeben sich für 'do'-, 'for'- und 'while'-Schleifen.
Ein Beispiel für die Anwendung der leeren Anweisung ist die folgende Zeitverzögerungsschleife:

```
for ( a=1; a < 1000; ++ a )
     ;
```

12.2.13 DIE 'PRINTF'-ANWEISUNG

Syntax:

```
printf(Formatanweisungen, Ausgabe);
```

Die durch die Formatanweisungen festgelegten Ausgaben werden auf den Bildschirm ausgegeben.

FORMATANWEISUNGEN BEI PRINTF()

Formatanweisungen stehen in Anführungszeichen und bestimmen den Ausgabetyp. Beachten Sie, daß bei mehreren auszugebenden Variablen in einer 'printf()'-Anweisung für jede Variable eine Formatanweisung stehen muß. Folgende Formatanweisungen sind möglich:

```
"%d" - Gibt eine Integerzahl aus.
"%u" - Ausgabe einer Unsigned Integerzahl.
"%f" - Ausgabe  einer Gleitkommazahl, einer 'float'-Variable.
"%e" - Ausgabe  einer 'float'-Variable in exponentieller Darstellung.
"%g" - Wählt  automatisch eine  "%e"- oder eine "%f"-Ausgabe abhängig
       von der Größe der Zahl.
"%c" - Ausgabe eines Character.
"%s" - Ausgabe eines Strings.
"%o" - Ausgabe einer Integerzahl in oktaler Darstellung.
"%x" - Ausgabe einer Unsigned Integerzahl in hexadezimaler Darstellung.
```

FORMATIERUNGEN DER AUSGABEN

Nach der "%"-Anweisung einer 'printf()'-Ausgabe kann eine eine Anweisung zur Ausgabeform folgen. Diese legt ein Ausgabefeld, die Genauigkeit und die Justierung einer Ausgabe fest.

Beispiele zur Formatierung:

```
"%8d"  - Ausgabe einer Integerzahl in einem Feld von acht
         Stellen.
"%-6f" - Ausgabe einer links-formatierten 'float'-Zahl.
"%6.2f"- Ausgbe einer 'float'-Variablen mit zwei
         Nachkommastellen und in einem gesamten Feld von
         sechs Stellen.
```

12.2.14 DIE 'RETURN'-ANWEISUNG

Syntax 1:

```
return;
```

Die 'return'-Anweisung bewirkt den direkten Sprung aus einer aufgerufenen Funktion zurück zu der Funktion, von der dieser Aufruf erfolgte. Am Ende einer Funktion, wenn ein Sprung zurück zu der Stelle, von der der Aufruf erfolgte, durchgeführt werden soll, kann die Anweisung 'return' entfallen.

Syntax 2:

```
return Ausdruck;
```

Der Wert des Ausdrucks wird zu der aufrufenden Funktion zurückgegeben.

12.2.15 DIE 'SCANF'-ANWEISUNG

Syntax:

```
scanf(Formatanweisungen, Argumente);
```

Die durch die Formatanweisungen festgelegten Argumente werden in das Programm übernommen und der Eingabevariablen zugeordnetn.

FORMATANWEISUNGEN FÜR SCANF()

Die 'scanf()'-Anweisung funktioniert ähnlich wie 'printf()'. Es ergeben sich jedoch die folgenden vier Unterschiede:

1. Die Argumente nach der Formatanweisung müssen Adressen darstellen. Möglich sind hier zum einen Adress-Operatoren ("&Variablen") oder Array-Namen.

Beispiel:

```
main()
{
  int x;
      char feld [100];
      scanf("%d %s", &x, feld)
          gemdos(0x1);
}
```

2. "%e" und "%f" werden gleich behandelt und nehmen 'float'-Variablen sowohl in normaler als auch in exponentieller Darstellung an.

3. Die Formatanweisung "%g" existiert nicht.

4. Es gibt eine spezielle "%h"-Option, die 'short int'-Variablen einließt.

12.2.16 DIE 'STRUCT'-ANWEISUNG

Syntax:

```
struct Name
   {
       Variablen Daklaration 1
       Variablen Deklaration 2
       ...
       Variablen Deklaration n
   };

    struct Name Variable 1, Variable 2, Variable n;
```

Zunächst wird die Strukturvariable 'Name' mit n-Variablen Deklarationen definiert. Die Variablentypen können beliebig

belegt werden. Vor der Verwendung der Struktur in einem Programm müssen die einzelnen Strukturvariablen noch als 'Variable 1' bis 'Variable n' definiert werden.

Der Zugriff auf die Strukturvariablen im Programm wird folgendermaßen gehandhabt:

An erster Stelle der Strukturvariablen steht die Variablendefinition 'Variable n'. Dann folgt ein Trennungspunkt und anschließend die einzelnen deklarierten Variablentypen 'Variablen Deklaration n'.

12.2.17 DIE 'SWITCH'-ANWEISUNG

Syntax:

```
switch( Ausdruck )
  {
       case Konstante 1:
            Anweisung 1
            Anweisung 2
            ...
            break;

       case Konstante 2:
            Anweisung 1
            Anweisung 2
            ...
            break;

       default:
            ...
  }
```

Der Wert des 'Ausdrucks' der Anweisung 'switch' wird mit den Werten der 'case'-Konstanten verglichen. Bei einer Übereinstimmung beider Werte werden die auf die entsprechende 'case'-Anweisung folgenden Befehle ausgeführt.

Es ist sinnvoll, die einmal ausgeführte 'case'-Anweisung über 'break' zu verlassen. Die bereits erörterte 'default'-Anweisung kann wahlweise in die 'switch'-Konstruktion integriert werden. Die auf 'default' folgenden Anweisungen werden ausgeführt, wenn keine 'case'-Bedingung erfüllt wurde.

12.2.18 DIE 'WHILE'-ANWEISUNG

Syntax:

```
while( Ausdruck )
    {
        Programm-Anweisungen
    }
```

Die Programm-Anweisungen werden solange wiederholt, wie der 'while'-Ausdruck wahr ist, d.h. solange der Wert des Ausdrucks ungleich null ist. Der Ausdruck kann auch durch eine Bedingung ersetzt werden.

12.3 VARIABLENTYPEN IN 'C'

Die Variablentypen lassen sich in zwei Kategorien einteilen, in Integer- und Gleitkommavariablen. Von diesen Grundtypen gibt es die folgenden Variationen, von denen einige jedoch nicht auf allen 'C'-Compilern vorhanden sind:

12.3.1 INTEGER-VARIABLEN:

char	Einzelner Characterwert.
int	Integerwert.
short int	Integerwert geringer Genauigkeit.
long int	Integerwert erhöhter Genauigkeit.
unsigned int	Positiver Integerwert, Werte können doppelt so groß sein wie bei normalen Integerzahlen.
unsignedshort	Positiver Integerwert geringerer Genauigkeit aber doppelter Größe.
unsigned long	Positiver Integerwert erhöhter Genauigkeit und doppelter Größe.

12.3.2 GLEITKOMMAVARIABLEN

float	Gleitkommazahl.
double float	Gleitkommazahl doppelter Genauigkeit.
long float	Wird wie 'double' behandelt.

12.4 OPERATOREN IN 'C'

In 'C' sind verglichen mit BASIC eine Fülle von Operatoren vorhanden. Die folgende Liste stellt die wichtigsten Operatoren zusammen. Die Liste ist nach der Priorität der einzelnen Operatoren geordnet. Dabei nimmt die Priorität zum Ende der Liste ab.

PRIORITÄTSSTUFE 1

()	Funktionsaufruf
[]	Array-Element
->	Pointer zum Struktur-Verweis-Operator
.	Struktur-Verweis-Operator

PRIORITÄTSSTUFE 2

-	Negatives Vorzeichen
++	Inkrement-Operator
--	Dekrement-Operator
!	Logische Umkehrung
*	Verweis-Operator
&	Adreß-Operator

PRIORITÄTSSTUFE 3

*	Multiplikations-Operator
/	Divisions-Operator
%	Modulo, Rest nach Divisions-Operator

PRIORITÄTSSTUFE 4

+	Additions-Operator
-	Subtraktions-Operator

PRIORITÄTSSTUFE 5

<<	Shift-nach-links-Operator
>>	Shift-nach-rechts-Operator

PRIORITÄTSSTUFE 6

```
<    Kleiner-Relation
<=   Kleiner-gleich-Relation
>    Größer-Relation
>=   Größer-gleich-Relation
```

PRIORITÄTSSTUFE 7

```
==   Gleich-Relation
!=   Ungleich-Relation
```

PRIORITÄTSSTUFE 8

```
&    UND-Verknüpfung von Bits
```

PRIORITÄTSSTUFE 9

```
^    Exklusive-ODER-Verknüpfung von Bits
```

PRIORITÄTSSTUFE 10

```
| (Pipe) Inclusive-ODER-Verknüpfung von Bits
```

PRIORITÄTSSTUFE 11

```
&&   Logische UND-Verknüpfung
```

PRIORITÄTSSTUFE 12

```
||   Logische ODER-Verknüpfung
```

PRIORITÄTSSTUFE 13

?: Bedingte Bewertung

Anhang A:

Tips & Tricks zu 'C' auf dem ATARI ST

Wenn Sie dieses Buch durchgearbeitet haben, was sollten Sie dann als nächstes tun?

Zunächst ist es sicher sehr sinnvoll, die gewonnenen 'C'-Kentnisse zu festigen. Dies geht am besten durch Programmierpraxis. Setzen Sie neue Algorithmen oder aber Ihre alten BASIC-Programme in 'C' um.

Dabei wird Ihnen der 'C'-Überblick des vorigen Kapitels sicher gute Dienste zum schnellen Nachschlagen bei der Anwendung von 'C'-Konstruktionen leisten. Wenn Sie genaue Informationen suchen, können Sie diese anhand des Inhaltsverzeichnisses sofort in den entsprechenden Abschnitten dieses Buches nachschlagen.

Sicher haben Sie bereits bemerkt, daß dieses Buch nicht nur ein Lehrbuch, sondern auch ein umfassendes Nachschlagewerk speziell für Sie als BASIC-Umsteiger darstellt. Der Schwerpunkt bei der Konzeption liegt dabei auf der Praxis. Das Buch sollte immer griffbereit neben Ihrem Computer liegen, wenn Sie in 'C' programmieren.

Wenn Sie Ihr Grundwissen so gefestigt haben, benötigen Sie mehr Informationen zu 'C'. Selbst in diesem umfangreichen 'C'-Buch konnten bei weitem nicht alle Elemente dieser komplexe Sprache erörtert werden.

Nachdem Sie einen Blick auf die sich im nächsten Abschnitt befindende 'C'-Bibliographie geworfen haben, werden Sie feststellen, daß es zwar sehr viele Einsteigerwerke für Anfänger ohne jede Computererfahrung, aber keine weiterführenden 'C'-Bücher gibt. Dem werden die Folgebände zu 'C' aus dem Hause DATA BECKER abhelfen.

Anhang B:

Literaturverzeichnis

DEUTSCHSPRACHIGE 'C' BÜCHER

Feuer, **"Das C Puzzle Buch"**, 180 Seiten, Hanser, München/ Pentice Hall Int, 1985

Hancock, **"C Programmierung - eine Einführung"**, 256 S., IWT, 1984

Illik, **"Erfolgreich programmieren mit C"**, Sybex, 1984

Kernighan, Ritchie, **"Programmieren in C"**, 275 S., Hanser, München, 1983

Lösch, Stanka, **"Die C-Sprache"**, 300 S., te-wi, München, 1985

Plum, **"Das C Lernbuch"**, Hanser, München/ Pentice Hall Int., 1985

Purdum, **"Einführung in C"**, 300 S. Markt & Technik, München, 1983

Schirmer, **"Die Programmiersprache C"**, 260 S., Hanser, München, 1985

ENGLISCHSPRACHIGE C-BÜCHER

AT & T Bell Labor., **"C Programmer's Handbook"**, 86 S., Prentice-Hall, 1985

Bean, **"The Illustrated C Programming Book"**, Prentice-Hall, 1985

Birns, Brown, Muster, **"UNIX for People"**, Prentice-Hall, 1984

Cooper, **"Graphics Programming in C"**, Sybex, 1985

Costales, **"C: From A to Z"**, Prentice-Hall, 1985

Feuer, **"The C Puzzle Book"**, Prentice-Hall, 198

Hendrix, **"The small C Handbook"**, Prentice-Hall, 1985

Harbison, Steele, **"C: A Reference Manual"**, Prentice-Hall, 1985

Hogan, **"The C Programmer's Handbook"**, Brady, 1984

Hunter, **"Understanding C"**, Sybex, 1985

Joyce, **"C by Example"**, Addison-Wesley, 1985

Kelley, **"A Book on C"**, Addison-Wesley, 1984

Kernighan, Ritchie, **"The C Programming Language"**, Prentice-Hall, 1978

Kochan, **"Programming in C"**, Hayden, 1983

Moore, **"Programming in C With a Bit of UNIX"**, Prentice-Hall, 1985

Plum, **"C Programming Guide Lines"**, Prentice-Hall, 1984

Plum, **"Learning to Programm in C"**, Prentice-Hall, 1983

Tondo, Gimpel, **"The C Answer Book"**, Prentice-Hall, 1985

Traister, **"Programming in C"**, Prentice-Hall, 1985

Traister, **"Programming Halo Graphics in C"**, Prentice-Hall, 1985

Tyler, **"Systems Programming in C"**, Sybex, 1985

Waite, Prata, Martin, **"C Primer Plus"**, Howard Sams, 1984

Wotman, Sidebottom, **"The C Programming Tutor"**, Brady Comp., 1984

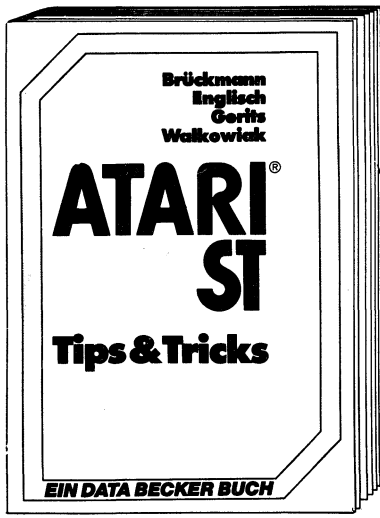
Zahn, **"C Notes: A Guide to the C Programming Language"**, Yourdon Press, 1979

Der neue ATARI ist eine Supermaschine!! Aber nur der richtige Einstieg garantiert den professionellen Umgang damit. Deshalb sollte dies Ihr erstes Buch sein. Eine leicht verständliche Einführung in Handhabung, Einsatz und Programmierung des ATARI ST: die Tastatur, die Maus, der Editor, der erste Befehl, das erste Programm, der Anschluß der Geräte u.v.m. Dieses Buch ist interessant geschrieben und ein Muß für jeden Einsteiger!

Lüers
ATARI ST für Einsteiger
ca. 250 Seiten, DM 29,—
ISBN 3-89011-152-1
Erscheint im Januar 1986

Das Informationspaket zum ATARI ST mit ausführlicher Hardwarebeschreibung, detaillierter Erläuterung der Schnittstellen: V.24, Expansion-Interface, MIDI-Interface, Aufbau von Grafiken, BIOS, GEM, wichtige Systemadressen und was man damit machen kann, die Funktionsweise der Maus. Unentbehrlich für's professionelle Arbeiten mit dem ATARI ST.

Gerits/Englisch/Brückmann
ATARI ST INTERN
ca. 400 Seiten, DM 69,—
ISBN 3-89011-119-X



Eine riesige Fundgrube faszinierender Tips & Tricks um Ihren ATARI ST voll auszunutzen! Von phantastischen Grafiken über raffinierte Programme in BASIC, Assembler und C bis hin zu fortgeschrittenen Anwendungsmöglichkeiten. Ein fantastisches Buch zu einem fantastischen Rechner!

Gerits/Englisch/Brückmann/ Walkowiak
ATARI ST Tips & Tricks
über 250 Seiten, DM 49,—
ISBN 3-89011-118-1

Sie haben den Einstieg auf dem ATARI ST geschafft? Dann werden Sie mit diesem Buch zum Profi. Aus dem Inhalt: Datenfluß- und Programmablaufpläne, fortgeschrittene Programmiertechniken, Menueerstellung, Grafikprogrammierung, mehrdimensionale Felder, Sortierroutinen, Dateiverwaltung und viele nützliche Utilities. So lernen Sie professionelles Programmieren!

Kampow
Das große BASIC-Buch zum ATARI ST
ca. 260 Seiten, DM 39,—
ISBN 3-89011-121-1

Den ATARI ST voll ausnutzen können Sie nur in Maschinensprache! Zahlensysteme, Bitmanipulation, der 68000 im ATARI ST, Registerverwendung, Struktur des Befehlssatzes, Programmstrukturen, Rekursion, Stacks, Prozeduren, Grundlagen der Assemblerprogrammierung Schritt für Schritt, Verwendung von Systemroutinen und Tips zum Einbinden von Assemblerroutinen in Hochsprachen. Eine hervorragend geschriebene Einführung!

Grohmann/Slibar/Seidler
ATARI ST Maschinensprache
über 220 Seiten, DM 39,—
ISBN 3-89011-120-3

Ein Buch für jeden, der unter GEM Programme erstellen will! Arbeiten mit der Maus, Icons, Virtual Device Interface, Application Environment Services und Graphics Device Operating System. Ein besonderer Schwerpunkt liegt im Einbinden von GEM-Routinen in BASIC und 68000-Assembler und der Programmierung in diesen Sprachen. GEM – das Betriebssystem der Zukunft!

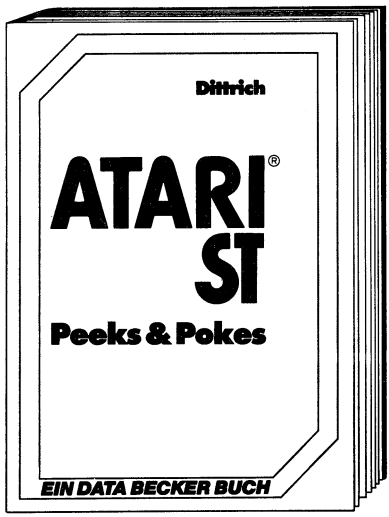
Szczepanowski/Günther
Das große GEM-Buch zum ATARI ST
ca. 350 Seiten, DM 49,—
ISBN 3-89011-125-4

Grafik und Sound auf dem ATARI ST. Ein Traum wird wahr! Grafikgrundlagen, Animationsgrafik, Funktionsdiagramme, 2-D/3-D Grafik, CAD, Soundgrundlagen und das MIDI-Interface sind nur einige Schwerpunkte dieses Buches. Alle Beispiele sind gründlich erklärt und mit vielen Beispielprogrammen verdeutlicht. Werden Sie zum Bildschirmkünstler und Computerdirigenten.

Walkowiak
ATARI ST Grafik & Sound
ca. 300 Seiten, DM 49,—
ISBN 3-89011-123-8

LOGO ist keineswegs nur eine Sprache für Kinder, sondern eröffnet viele interessante Bereiche wie z. B.: Rechnen mit LOGO, Grafikprogrammierung, Wörter- und Listenverarbeitung, Prozeduren, Rekursionen, Sortierroutinen, Maskengenerator, Datenstrukturen und Künstliche Intelligenz. Mit LOGO können Sie schwierige und komplexe Probleme oft leichter lösen als mit anderen Programmiersprachen!

Dr. Sauer
Das LOGO-Trainingsbuch zum ATARI ST
ca. 250 Seiten, DM 49,—
ISBN 3-89011-122-X
Erscheint Januar 1986



Schlagen Sie dem Betriebssystem Ihres ATARI ST ein Schnippchen. Wie? Mit PEEKS & POKES natürlich! Dieses Buch erklärt Ihnen leichtverständlichh den Umgang damit. Mit einer riesigen Anzahl wichtiger POKES und ihren Anwendungsmöglichkeiten. Dabei wird der Aufbau Ihres ST's prima erklärt: Betriebssystem, Interpreter, Zeropage, Pointer und Stacks sind nur einige Stichworte dazu. Der erste Schritt hin zur Maschinensprache!

PEEKS & POKES zum ATARI ST
ca. 250 Seiten, DM 29,—
ISBN 3-89011-148-3

Kein Programmierer, der die Vorteile des 68000-Prozessors nutzen will, sollte auf dieses Handbuch verzichten. Sie finden detailliertes Sachwissen, anschaulich dargestellt, zur Technik und Programmierung: Entwicklung des 68000, Aufbau, Signal- und Busbeschreibung, Peripheriebausteine, Befehlssatz, Programmierbeispiele, Vergleich mit anderen 16-Bit-Prozessoren, weitere Prozessoren der Familie u.v.m. Ein Buch für echte Computerfreaks!

Grohmann/Eichler
Das Prozessorbuch zum 68000
516 Seiten, DM 59,—
ISBN 3-89011-094-0

DAS STEHT DRIN:

Wenn Sie die Grundelemente von BASIC kennen, ist dieses Buch das richtige für Sie, um schnell in "C" einzusteigen! Ausgehend von einfachen BASIC-Programmen wird Schritt für Schritt der entsprechende "C"-Code entwickelt und didaktisch ausführlich erläutert. Durch diese Methode lernen Sie die Grundstrukturen der Supersprache "C" an einem Tag!

Aus dem Inhalt:

- Entwicklung, Anwendung und Stellenwert von "C"
- Erste Schritte für BASIC-Umsteiger
- Die Funktionen und Textausgabe auf dem Bildschirm
- Das Programmformat
- Variablendeklaration
- Formatanweisung
- Schleifen und Kommentare
- Die Dateneingabe
- Die Arithmetik in "C"
- Kontrollstrukturen
- Datentypen in "C"
- "C"-Pointer
- Arrays
- Charakteristische Fehler von BASIC-Umsteigern

UND GESCHRIEBEN HAT DIESES BUCH:

Olaf Hartwig, "C"-Experte und Autor zahlreicher Bücher, stieg als einer der ersten in die Welt der Microcomputer ein. Er besitzt einschlägige Erfahrungen auf allen gängigen Home- und PC-Computern und wurde für seine herausragenden Programmierleistungen vom Bundesminister für Forschung und Technologie, Dr. Heinz Riesenhuber, ausgezeichnet.

ISBN 3-89011-147-5