Th. Biedermann

Grundlagen der Programmierung in C

Eine kurzgefaßte Einführung in die Sprachelemente der Sprache 'C'

1 Programmstruktur

1.1 Reihenfolge der Deklarationen in einem C-Programm

Ähnlich wie in der Sprache Pascal ist bei der Anordnung der einzelnen Teile eines Programmes in der Regel eine bestimmte Reihenfolge einzuhalten. Da es sich bei C um eine prinzipiell modular ausgerichtete Sprache handelt, existiert eine größere Anzahl verschiedener vordefinierter Bestandteile, deren Bedeutung für das Programm zum Teil entscheidend durch ihre Plazierung im Programmkontext beeinflußt wird. Um Konfusion zu vermeiden, sollte man sich für eine bestimmte Rangfolge entscheiden und diese beibehalten, solange keine anderen Gründe eine Abweichung erforderlich machen. Üblicherweise wird folgende Anordnung verwendet:

Programmteil	Erläuterung
#include <*dateiname***.h>**	Dateien mit der Endung *.h* enthalten üblicherweise vordefinierte Variablentypen oder Funktionsdeklarationen
#define *param1 param2*	Der Quelltexteintrag param1 wird im Programmtext vor dem Kompilieren durch den Text param2 ersetzt
typedef *typ neuer_type;*	Mit typedef können beliebige Datentypen benannt werden, um den Typ bestimmter Variablen herauszuheben
struct *struktur_definition;*	Eine Struktur faßt unterschiedliche Datentypen zu einem einheitlichen Gebilde zusammen
Funktionendeklarationen;	Hier können Funktionen deklariert werden, die sich in anderen Programmteilen befinden
Externe Variable;	Hier deklarierte Variable stehen allen Modulen eines Programmes zur Verfügung
main (...) { *Lokale Variable;* *Anweisungsteil;* }	Es handelt sich um ein Hauptprogramm, erkennbar daran, daß es ein Modul mit dem Namen *main* gibt Diese Variablen sind lokal zu diesem Modul (Ein Modul mit dem Namen *main* darf nur einmal pro Hauptprogramm auftreten, auch wenn sich dieses aus mehreren Teilprogrammen zusammensetzt)
Funktionsname(...) { *Lokale Variable;* *Anweisungsteil;* }	Hier (oder in anderen Teilprogrammen) werden alle Routinen deklariert, die nicht in einer Standardbibliothek enthalten sind Jedes Modul kann lokale Variablen benutzen. Anzahl, Größe und Verschachtelungstiefe der Subroutinen ist kaum Beschränkungen unterworfen, die Reihenfolge spielt keine Rolle.
...	

(Wahlfreie bzw. lediglich beschreibende Begriffe in der Spalte Programmteil sind *kursiv* gesetzt, während C-spezifische Bezeichner **fett** gedruckt sind.)

Im einfachsten Fall kann ein Programm lediglich aus der Funktion *main()* bestehen, wie z.B. das folgende Beispiel zeigt:

```
main()
{
  printf("Hello World \n");
}
```

1.2 Aufbau einer Funktion

Wie im vorhergehenden Kapitel angedeutet, besteht ein C-Programm aus Deklarationen, Definitionen und Anweisungen. Erstere werden vor allem dann benötigt, wenn auf bereits in anderen Modulen vordefinierte Strukturen und Funktionen verwendet werden sollen, während letztere den eigentlichen Algorithmus repräsentieren, den das Programm abarbeiten soll.
Grundsätzlich ist jeder abgeschlossene Teil eines C-Programmes als eine Funktion - oder, im weiteren Sinne, als ein Modul - aufzufassen, das eine bestimmte Aufgabe übernimmt. Der Zugriff auf eine solche Funktion erfolgt über ihren Namen, dessen Bildung bestimmten Regeln unterworfen ist und einmalig in einem Programm sein muß. Die Übergabe von Daten in Form von Variablen ist wahlfrei und kann auf die verschiedensten Arten erfolgen. Da jedes Modul eine Funktion darstellt, kann ein Wert zurückgegeben werden, eine Unterscheidung zwischen Prozedur und Funktion - wie in Pascal - ist nicht nötig.
Die Definition einer Funktion erfolgt nach einem Schema, das je nach Bedarf stellenweise vereinfacht werden kann.

Eine Funktion besteht aus einem *Funktionskopf* und einem *Anweisungsteil*, dem gegebenenfalls eine lokaler *Deklarationsteil* vorausgehen kann.

Der *Funktionskopf* erlaubt den Aufruf der Funktion durch ihren Namen und definiert die Liste der eventuell zu übergebenden Variablen:
Nach einer Typfestlegung (die erst zu einem späteren Zeitpunkt erläutert wird) folgt der Funktionsname. Dieser muß in den ersten 8 Zeichen einzigartig sein, wobei Groß- und Kleinschreibung unterschieden werden. Als erstes Zeichen muß ein Buchstabe oder der Unterstrich _ verwendet werden, als weitere Zeichen sind auch Ziffern erlaubt. Der Name darf insgesamt natürlich länger als 8 Zeichen sein, aber die zusätzlichen Zeichen sind nur noch für den Benutzer von Belang, während sie vom Compiler ignoriert werden. Hinter dem Funktionsnamen folgt nach einer öffnenden runden Klammer eine Liste der zu übergebenden Variablen, die durch Kommata getrennt und mit einer schließenden runden Klammer beendet wird. Die Variablenliste darf entfallen, nicht aber die beiden Klammern. Nach dieser Definition der Funktionskopfes darf kein Semikolon stehen!
Wird in den runden Klammern eine nicht leere Liste von Variablen übergeben, müssen anschließend deren Typen deklariert werden, diese Typdeklarationen sind zeilenweise durch ein Semikolon abzuschließen.

Der Anweisungsteil legt die Aktionen fest, die von dieser Funktion in Abhängigkeit von und je bei Bedarf auch mit den übergebenen Variablen (oder global definierten Variablen) durchgeführt werden sollen. Sind dazu weitere lokale Variablen nötig, müssen diese vor der ersten ausführbaren Anweisung definiert und deklariert werden. Der Anweisungsteil wird durch geschweifte Klammern eingeschlossen, sie haben in etwa die Bedeutung eines durch BEGIN und END eingeschlossenen Blockes in Pascal.

Im folgenden Beispiel einer einfachen Funktion sind alle eben aufgeführten Elemente enthalten, dabei wurden die Teile fett und kursiv gedruckt, die unbedingt vorhanden sein müssen:

```
int fehler(f_nr)
short f_nr;
{
  char ch;

  printf("In dem Programm ist der Fehler #%d aufgetreten.\n",f_nr);
  printf("Drücken Sie bitte eine Taste ...\n");
  ch = getch();
}
```

1.3 Aufbau eines Programmes

Ein C-Programm besteht aus mindestens einer Funktion mit dem Namen *main()* und je nach Bedarf unterschiedlich vielen weiteren Funktionen, die von *main* oder einer anderen Funktion verwendet werden. Bei der Erstellung der einzelnen Module gibt es verschiedene Möglichkeiten, diese in Dateien anzuordnen, wobei das Prinzip der Modularisierung einen entscheidenden Einfluß hat.

Eine Datei mit nur einem Anweisungsteil
Am wenigsten modularisiert ist ein Programm, das lediglich aus der Funktion*main* besteht. Während das für einen sehr einfachen Algorithmus noch akzeptabel erscheint, ist es bei komplexeren Problemstellungen nur in Ausnahmefällen sinnvoll.

Eine Datei mit mehreren Anweisungsteilen
Ebenfalls nur als wenig modularisiert muß man ein Programm bezeichnen, das neben der Funktion*main* zwar noch weitere Funktionen enthält, die von*main* aufgerufen werden, welche aber eher mit dem Ziel einer Strukturierung und ökonomischen Programmgestaltung angelegt wurden. Dies ist sicherlich nur bei kleineren und überschaubaren Problemstellungen sinnvoll, die in ihrer Spezialisierung den Aufwand für eine universellere Verwendbarkeit als unangemessen erscheinen lassen.

Mehrere Dateien mit mehreren Anweisungsteilen
Verlagert man zusammengehörende Anweisungsteile jeweils in eigene Dateien, die für sich kompilierbar sind, so erhält man zwangsläufig eine Modularisierung, die z.B. eine gleichzeitige Erstellung der einzelnen Programmteile durch verschiedene Programmierer erlaubt. Zwar sind diese Module für sich allein in der Regel nicht lauffähig, da sie keine Funktion mit dem Namen *main* enthalten (dürfen), andererseits kann ihre Funktionsfähigkeit leicht durch geeignete Testprogramme überprüft werden. Ist eine Funktionsgruppe vollständig und fehlerfrei programmiert, kann sie allen anderen zur Verfügung gestellt werden. Durch die Vorgehensweise des C-Kompilers erübrigt sich außerdem eine ständige Neuübersetzung des Programmcodes, was die Übersetzungszeiten größerer Projekte drastisch reduziert.

1.4 Erstellung eines ausführbaren Programmcodes

Um aus einem Quelltext ein ausführbares Programm zu erhalten, geht der C-Compiler in verschiedenen Stufen vor, die bei einem integrierten System wie QuickC automatisch nacheinander ablaufen.

Pre-Prozessor
Im ersten Schritt wird der vom Benutzer geschriebene Quellcode (der in der Regel im ASCII-Format vorliegen muß), nach bestimmten Schlüsselwörtern durchsucht, die den anschließenden Kompiliervorgang beeinflussen können. Hierzu gehören z.B. die Direktiven *#define*, mit der eine bestimmte Zeichenfolge durch eine andere ersetzt werden kann und die Direktive*#include*, mit der in einer anderen ASCII-Datei enthaltene Textzeilen an der gewünschten Stelle eingefügt werden.

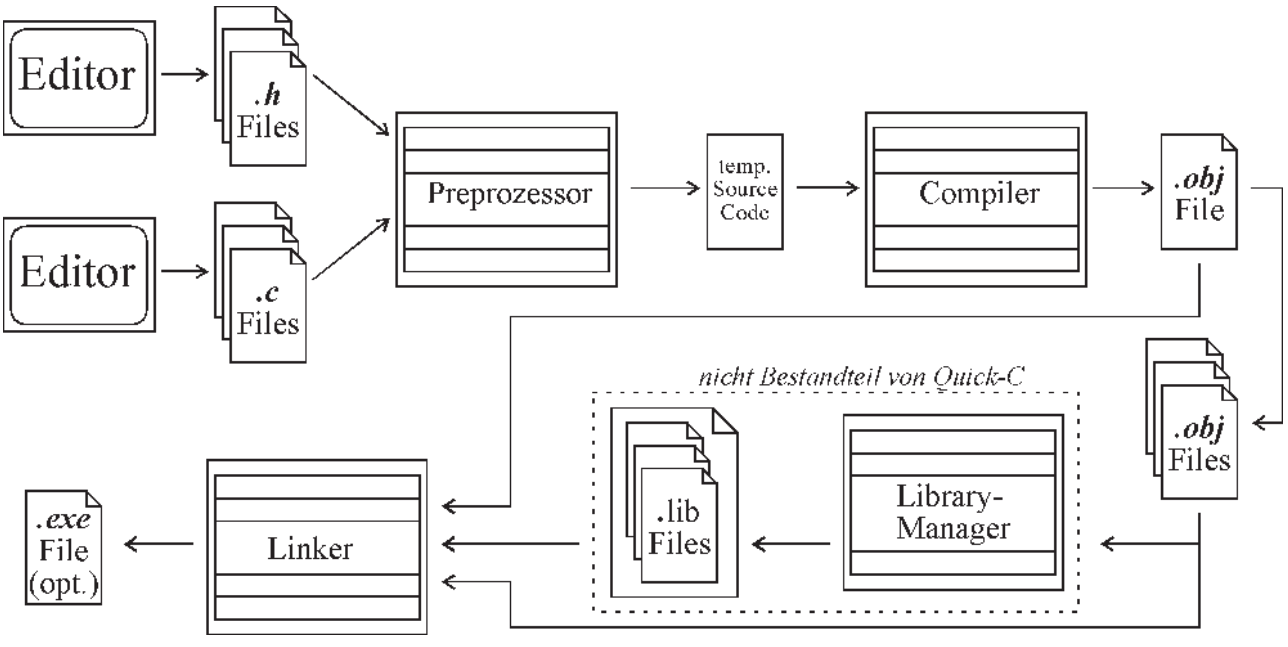
Kompiler-Lauf, Pass 1
Im ersten Kompiler-Lauf werden die symbolischen Bezeichner und ihre Deklaration im Programmkontext überprüft und tabellarisch erfaßt, um einerseits Querverweise zu erzeugen, andererseits aber unzulässige Wiederholungen bereits benutzter Bezeichner festzustellen. Identifizierte Anweisungsteile werden hierbei bereits in einen ausführbaren Code übersetzt, wobei jedoch die Speicherorte von Funktionen und Variablen noch offen bleiben.

Kompiler-Lauf, Pass 2
Im zweiten Kompiler-Lauf werden alle in Anweisungsteilen deklarierten Funktionen, Variablen und Konstanten lokalisiert und relativ zum ausführbaren Code einem Speicherplatz zugewiesen. Die Namen aller auch von außerhalb dieses Moduls zugänglichen Bezeichner und ihre Adressen innerhalb des Moduls werden in einem besonderen Format als Kopf einer Datei eingetragen, die man als Objekt-Datei bezeichnet und in der Regel die Datei-Endung .OBJ trägt. Hinter diesem Kopf folgen die Maschinencodes und die Speicherstellen der definierten Variablen, die durch die Anweisungsteile der einzelnen Funktionen definiert wurden.

Linker-Lauf
Anschließend wird durch einen Linker (dtsch: Verbinder) in allen angemeldeten Modulen nach einer Funktion gesucht, die den Namen *main* trägt. Wird diese nicht gefunden, bricht der Linker die weitere Abarbeitung ab, da er ohne diese Funktion kein Hauptprogramm erstellen kann. Ansonsten überprüft er die Köpfe aller zugeordneten Objekt-Dateien, ob alle darin enthaltenen Verweise auf außerhalb eines Moduls liegenden Adressen in einem anderen Modul definiert sind. Werden diese in keinem der anderen Module gefunden, werden auf die gleiche Weise alle vordefinierten und zugänglichen Bibliotheken durchsucht. Bleiben anschließend immer noch unaufgelöste Verweise übrig (unresolved externals), kann kein ausführbares Programm erstellt werden und der Linker bricht ab. Ansonsten entsteht eine Datei mit der Datei-Endung .EXE, die beliebig oft ausgeführt werden kann.

Besonderer Hinweis für Quick-C: Quick-C erlaubt die Kompilierung einer ausführbaren Datei wahlweise im Speicher oder auf der Platte, eine OBJ- und eine EXE-Datei wird jedoch nur im zweiten Fall erstellt.

Vorgehensweise des Compilers
Die folgende schematische Darstellung stellt die Reihenfolge der Abarbeitung bzw. Erzeugung der Dateien zur Erstellung eines ausführbaren Programmes dar. Dabei können die einzelnen Schritte innerhalb eines Programmes integriert (wie z.B. in Quick-C) oder auf einzelne, dann meistens erheblich leistungsfähigere Programme verteilt sein (wie z.B. bei MS-C). Im letzteren Fall ist auch die Kombination von Programm-Modulen (OBJ-Files) verschiedener Programmiersprachen möglich. In fast allen Stufen der Programmerstellung können (im Schema nicht aufgeführt) automatisch unterschiedliche Protokolldateien angelegt werden, die zur Fehlersuche und -analyse sowie zur Programm- und Datenoptimierung herangezogen werden können.



2. Sprachelemente

2.1 Datentypen

Verschiedene Datentypen sind in C vordefiniert, wobei die Implementation zum Teil maschinenabhängig ist. Die folgenden Angaben beziehen sich auf das üblicherweise auf PC's verwendete Schema.Wenn in einigen Fällen verschiedene Namen für ein und den selben Datentyp möglich sind, wurde der kürzere Name angegeben. Hierbei ist zu beachten, daß alle Datentypen einen numerischen Wertebereich definieren: Zeichen (Characters) werden ausschließlich über ihren ASCII-Code repräsentiert.

Name		Memory	Wertebereich Bemerkung
char	1 byte	-128 ... 127	vorwiegend für ASCII-Zeichen
int	2 byte	-32768 ... 32767	ganze Zahl mit Vorzeichen
short	2 byte *	-32768 ... 32767	ganze Zahl mit Vorzeichen
long	4 byte	-2147483648 ... 2147483647	große ganze Zahl mit Vorzeichen
unsigned char	1 byte	0 ... 255	vorzeichenlose 8 Bit Zahl
unsigned	2 byte	0 ... 65535	vorzeichenlose 16 Bit Zahl
unsigned short	2 byte *	0 ... 65535	vorzeichenlose 16 Bit Zahl
unsigned long	4 byte	0 ... 4294967295	vorzeichenlose 32 Bit Zahl
enum	2 byte	0 ... 65535	vorzeichenlose 16 Bit Zahl
float	4 byte	$\sim 3.4 \cdot 10^{-38}$ ... $3.4 \cdot 10^{38}$	reelle Zahl mit 7 sign. Ziffern
double	8 byte	$\sim 1.7 \cdot 10^{-308}$ ... $1.7 \cdot 10^{-308}$	reelle Zahl mit 15 sign. Ziffern
long double	8 byte *	$\sim 1.7 \cdot 10^{-308}$ ... $1.7 \cdot 10^{-308}$	reelle Zahl mit 15 sign. Ziffern

(die mit * gekennzeichneten Datentypen sind maschinenabhängig und somit nicht auf andere Rechnersysteme übertragbar)

2.2 Operatoren

Es existiert eine Vielzahl von Operatoren, die je nach Kontext eine unterschiedliche Bedeutung haben können. In der folgenden Liste sind sie deshalb einem bestimmten Level zugeordnet. Operatoren eines kleineren Levels haben Vorrang vor höheren Levels, während für Operatoren gleichen Levels kein Vorrang definiert ist.

Level	Operator	Bedeutung	Beispiel
1	( )	Funktion	main**(**argc,argv**)**
	[ ]	Array	feld**[**3**]**
	.	Komponente	register**.**eintrag
	->	Komponente	register->eintrag
2	-	Arithmetisches Minus	-3 {negative Zahl}
	~	Bitweises Komplement	~254 {ergibt 1}
	!	Logische Verneinung	**!**true {= false}
	*	Indirekte Adresse	***string**
	&	Adressoperator	**&**zahl
	++	Inkrementieren	**++**wert {wert := wert + 1}
	--	Dekrementieren	**--**wert {wert := wert - 1}
	sizeof	Größe in Byte	**sizeof**(int)
	(type)	Typ-Zuweisung	char_typ = **(char)** int_typ

Level	Operator	Bedeutung	Beispiel
3	*	Multiplikation	x = 3 * y;
	/	Division	y = x / 3;
	%	Rest	z = 8 % 3; { = 2}
4	+	Addition	x = 3 + y;
	-	Subtraktion	x = 3 - y;
5	<<	Bitweise links schieben	8 << 2 { = 32}
	>>	Bitweise rechts schieben	8 >> 2 { = 2}
6	<	Kleiner als	x < 5
	>	Größer als	x > 3
	<=	Kleiner oder gleich als	x <= 5
	>=	Größer oder gleich als	x >= 3
7	==	Gleichheit	4 == (2*2)
	!=	Ungleichheit	5 != 4
8	&	Bitweise UND-Verknüpfung	3 & 7 = 7
9	^	Bitweise exclusiv ODER	3 ^ 7 = 4
10	|	Bitweise inclusiv ODER	3 | 4 = 7
11	&&	Logisches UND	true && false = false
12	||	Logisches ODER	true || false = true
13	a1?a2:a3	Konditionale Aussage	x == 5 ? x=0 : x=3 {wenn x = 5 ist, dann x := 0, sonst x := 3 setzen}
14	=	Zuweisung	x = 5 { x nimmt den Wert 5 an}
	+=	erhöhen und zuweisen	x += 3 {x wird um 3 erhöht}
	-=	erniedrigen und zuweisen	x -= 4 {x wird um 4 erniedrigt}
	*=	multiplizieren und zuweisen	x *= 2 {x wird verdoppelt}
	/=	dividieren und zuweisen	x /= 2 {x wird halbiert}
	%=	Rest bilden und zuweisen	x %= 3
	>>=	Bitweise verschieben u. zuweisen	x >>= 2 {schieben um 2 Bit nach rechts}
	<<=	Bitweise verschieben u. zuweisen	x <<= 2 {schieben um 2 Bit nach links}
	&=	Bitweise AND-Verkn. u. zuweisen	x &= 127 {8. Bit löschen}
	|=	Bitweise OR-Verkn. u. zuweisen	x |= 128 {8. Bit setzen}
	^=	Bitweise XOR-Verkn. u. zuweisen	x ^= 128 {8. Bit invertieren}
15	,	Mehrfach-Zuweisung	for (i = 0 , j = 1 ; ...)

2.3 Steuercodes (Escape-Sequenzen)

Zur formatierten Ausgabe von Texten und Zeichen auf dem Bildschirm sind einige Sonderzeichen definiert, die durch ein "Escape-Zeichen" eingeleitet werden, welches in C durch einen Rückwärts-Schrägstrich \ (Backslash) repräsentiert wird:

Zeichen	Bedeutung	Zeichen	Bedeutung	Zeichen	Bedeutung
\a	Bell (Alarm)	\v	Vertikal Tab.	\'	einf. Anführungsz.
\n	New Line	\t	Horiz. Tab.	\"	dopp. Anführungsz.
\b	Backspace	\f	Formfeed	\ddd	ASCII-Char (octal)
\r	Carriage Return	\\	Backslash	\xddd	ASCII-Char (hexadez.)

2.4 Kontrollstrukturen

Wie jede Programmiersprache benötigt C einen Satz von Kontrollstrukturen, die es erlauben, Bedingungen zu formulieren und die Abarbeitung bestimmter Anweisunungsteile vom aktullen Wert einer Variablen abhängig zu machen oder Wiederholungsstrukturen zu bilden.

Die folgende Auflistung führt die 11 definierten Kontrollstrukturen in alphabetischer Reihenfolge mit jeweils einer kurzen Erläuterung und einem kurzen Programmbeispiel auf.

2.4.1 break;

Dieses Kommando beendet die Abarbeitung des aktuellen Anweisungsteils eines do-, for-, switch- oder while-Statements, indem es bis zur nächsten schließenden geschweiften Klammer springt und dort die Bearbeitung fortsetzt.

```
for (i = 0; i < 10; i++)
  {
    ...
    if (fehler) break; /* Abbruch der Schleife unabhängig von i /*
    ...
  }
```

2.4.2 { } (geschweifte Klammern)

Mit diesen Klammern wird der Compiler angewiesen, Speicherplatz für die in dieser Einheit deklarierten Variablen und die Befehlscodes bereitzustellen, soweit es sich nicht um bereits anderweitig lokalisierte Elemente handelt. Damit geht die Bedeutung der geschweiften Klammern über die Begin-End-Konstruktion von Pascal hinaus, da in Pascal Variablen nur außerhalb dieser Struktur deklariert werden können.

```
{
  (deklarationen)
  ...
  (anweisungen)
  ...
}
```

2.4.3 continue;

Hiermit wird der Anweisungsteil eines do-, for- oder while-Statements vorzeitig abgebrochen und die nächste Iteration aufgerufen.

```
for (i = 0; i < 10; i++)
{
  ...
  if (ueberspringen) continue;    /* nächster Schleifendurchlauf */
  ...
}
```

*2.4.4 **do** statement; **while** (expression);*

Die Anweisung*statement*(die auch aus einem durch geschweifte Klammern bestehenden Anweisungsteil bestehen kann), wird so lange ausgeführt, bis die in *expression* genannte Bedingung falsch wird (bzw. den Wert 0 annimmt). Dies entspricht der Repeat-Until-Konstruktion in Pascal.

```
do
  {
    printf("Durchlauf #%d",i++);  /* wird mind. 1 x ausgeführt! */
  }
while (i < 10);
```

2.4.5 for (init; condition; loop) statement;

Die Anweisung *statement*, die auch einen durch geschweifte Klammern gebildeten Anweisungsteil umfassen kann, wird mit dem in *init* definierten Startwert begonnen und solange durchlaufen, wie die in *condition* genannte Bedingung erfüllt ist (d.h., einen Wert ungleich 0 annimmt). Dabei wird bei jedem Durchlauf die in *loop* definierte Veränderung der dort spezifizierten Variablen vorgenommen. Diese Struktur bietet erheblich mehr Kontrollmöglichkeiten als die entsprechende Pascal-Struktur, da in den drei Schleifenkontrollen unterschiedliche Variablen und auch Funktionen benutzt werden können.

```
for (i = 0, j = 1, k = 0; j < 10; k++)
  {
    i = j + k;   /* i wird bei 0 beginnend um j und k erhöht    */
    j = k - 1;   /* wenn k = 11, wird j = 10 -> Ende            */
  }
```

2.4.6 goto name ... name: statement;

Die Programmausführung wird direkt an der Stelle fortgesetzt, die durch *name:* im Anweisungsteil spezifiziert ist. Da diese Anweisung die sequentielle Struktur des Programmes durchbricht, sollte sie **außer bei Programmtests** nicht angewendet werden.

```
{
  ...
  if (bedingung) goto marke1;
  ...
  return(!fehler);
marke1: return(fehler);
}
```

2.4.7 if (expression) statement1; [else statement2;]

Wenn der in *expression* aufgeführte Ausdruck wahr ist (d.h., einen Wert ungleich 0 annimmt), wird die Anweisung *statement1* ausgeführt. Fehlt der *else*-Teil, wird mit der nächsten Anweisung fortgefahren, ansonsten wird die nach *else* als *statement2* aufgeführte Anweisung ausgeführt.

```
if ((ch = getch()) != 32)
  {
    printf("Unzulässige Eingabe: (Taste...)");
    getch();
  }
else printf("Danke für die Mitarbeit! \n");
```

2.4.8 ; (Semikolon)

Das Semikolon schließt jeden Befehl ab. Ein Semikolon ohne vorausgehenden Befehl ist eine leere Anweisung.

```
  printf("\n");
  ;              /* Eigentlich ohne Belang, da hier auch eine   */
                 /* Leerzeile hätte eingefügt werden dürfen!    */
  ...
```

2.4.9 return (expression)

Dieses Kommando beendet den aktuellen Anweisungsteil und übergibt die Programmkontrolle sowie den in *expression* definierten Wert an das übergeordnete Programm, das diesen Anweisungsteil aufgerufen hatte.

```
  ...
  if (i >= 0) return(1); else return(-1); /* Vorzeichen */
}
```

*2.4.10 **switch** (expression) { declaration ... **case** expression: statement1; **[default:** statement2] }*

Diese Anweisung entspricht der *Case*-Anweisung von Pascal mit einigen Unterschieden in der Reihenfolge der Abarbeitung der Anweisungsteile zu den einzelnen Einträgen. Die *switch-expression* legt den Wert einer Variablen oder eines Ausdrucks fest, der von den einzelnen *case-expressions* überprüft wird. Bei Übereinstimmung werden die auf die *case-expression* folgenden Anweisungen ausgeführt, bis der Befehl *break* oder die die *switch*-Anweisung schließende geschweifte Klammer gefunden wird. Ist ein (wahlfreies) *default-statement* angegeben, wird dieses dann ausgeführt, wenn keine der *case-expressions* mit der *switch-expression* übereinstimmt.

```
switch (ch = getch())
  {
    int i;
    case '0':
    case '1':
              i = ch - '0';
              printf("Zahl %d ist kleiner als 2 ",i);
    case '2':
              i = ch - '0';
              printf("Zahl %d ist kleiner oder gleich 2",i);
              break;
    default :
              printf("Eingabe ist keine Zahl oder größer als 2");
              return (ch);
  }
...
```

Wurde die Taste '0' oder '1' gedrückt, erscheint die Bildschirmmeldung "Zahl 0 (oder 1) ist kleiner als 2 Zahl 0 (oder 2) ist kleiner oder gleich 2", denn die *case-expressions* werden wie Sprungmarken behandelt, d.h., in den beiden genannten Fällen erfolgt der Einsprung in die Anweisungsfolge, die mit der ersten *printf*-Anweisung beginnt - die weiteren Anweisungen werden ausgeführt, bis das *break-statement* erreicht ist. Wurde die Taste '2' gedrückt, wird der Text "Zahl 2 ist kleiner oder gleich 2" ausgegeben, danach greift bereits die *break*-Anweisung. Bei jeder anderen Eingabe erfolgt die Textausgabe "Eingabe ist keine Zahl oder größer als 2" und die Funktion wird wegen des *return-statements* mit dem eingegebenen Zeichen als Rückgabewert verlassen, während bei den vorgenannten Beispielen die Funktion mit den auf die *switch*-Anweisung folgenden Befehlen fortgesetzt würde.
Werden innerhalb der *switch*-Konstruktion lokale Variablen benötigt, können diese nach der öffnenden geschweiften Klammer und der ersten *case-expression* angeordnet werden, wie im oberen Beispiel gezeigt.

*2.4.11 **while** (expression) statement;*

Diese Anweisung entspricht der gleichnamigen Anweisung in Pascal. Die Anweisung (oder der in geschweiften Klammern gesetzte Anweisungsteil) *statement* wird so lange ausgeführt, wie der Wert von *expression* ungleich 0 ist.

```
scanf("%d",&i);
while (i)
  {
    printf("&d ",i);
    i = i / 2;
  }
```

Da in C der boolesche Ausdruck *false* gleichbedeutend mit dem Variablenwert 0 ist, genügt hier die Angabe der Variablen allein (statt des schreibintensiveren Terms *i = 0*).

2.5 expressions - Ausdrücke

Als *expression* bezeichnet man jeden gültigen Ausdruck, der einen eindeutigen Wert annimmt. Der einfachste Ausdruck in Pascal besteht in dem reservierten Wert *false* oder *true*, der auch als Aussage, z.B. in der Form x = 3 formuliert werden kann (*true*, wenn x = 3, sonst *false*). Um in C einen Unterschied zwischen der Zuweisung (x nimmt den Wert 3 an) und dem zu bewertenden Ausdruck (hat x den Wert 3?) zu bewerkstelligen, wird die Zuweisung mit dem einfachen Gleichheitszeichen (x = 3) entsprechend der Zuweisung in Pascal (x := 3) vorgenommen, während der Vergleich konsequent mit doppelten Relationszeichen (z.B. x == 3; x >= 3; x != 3) dargestellt wird. Im letzteren Fall ist Pascal hier inkonsequenter (x = 3, aber x <> 3).
In C kann jede Zuweisung gleichzeitig einen Ausdruck darstellen, indem ein Ausdruck automatisch *true* wird, wenn er einen Wert ungleich 0 annimmt. Der Unterschied ist nur aus dem Kontext zu entnehmen, wie das folgende Beispiel zeigt:

```
(A)    i = getch()
```

ist in dieser Form zunächst eine Zuweisung. Der Wert von *i* soll nun in eine *if*-Bedingung eingehen:

```
(B)    if (i != 0) ...
```

In diesem Fall wird die Anweisung ausgeführt, wenn *i* einen Wert ungleich 0 annimmt. Da in C dieser Wert als *true* definiert ist, genügt hier aber auch schon die Schreibweise

```
(C)    if (!i) ...
```

da beim Prüfen der Bedingung ohnehin der Wert von *i* auf 0 getestet werden muß.
In C ist es nun außerdem möglich, die Zuweisung von *i* als Parameter des Ausdrucks der *if*-Bedingung anzugeben, also in der Form:

```
(D)    if ((i = getch()) != 0) ...
```

Man beachte, daß in (D) die Angabe der Variablen aus (B) durch den Ausdruck (A) ersetzt wurde, der die Variable *und* deren Zuweisung beinhaltet. Berücksichtigt man, daß der in (D) geklammerte Ausdruck ohnehin von C auf den Wert 0 geprüft werden muß, damit ein Vergleich mit der Konstanten 0 möglich ist, liegt es nahe, die in (C) angegebene Notierung mit der in (D) gegebenen Vereinfachung zu verknüpfen, man erhält

```
(E)    if (!(i = getch())) ...
```

was ebenfalls zur Ausführung der Anweisung führt, wenn für *i* ein Wert ungleich 0 eingegeben wird.
Folglich ist die Anweisungsfolge

```
i = getch();
if (i != 0) ...
```

gleichbedeutend mit der kompakteren Anweisung

```
if (!(i = getch())) ...
```

*2.5.1 expression1 **?** statement1 **:** statement2*

Da bedingte Anweisungen der Form ***if*** *expression* ***then*** *statement1* ***else*** *statement2* besonders häufig sind, existiert in C eine Kurzschreibweise für diese Kontrollstruktur. Gemäß dem oben angebenen Beispiel für die Zuordnung und anschließende Überprüfung einer Variablen ergibt sich folgende Notierung:

```
(i = getch()) ? funktion1() : funktion2();
```

hierbei sind *funktion1()* und *funktion2()* zwei beliebige Funktionen oder Anweisungen. *funktion1()* wird ausgeführt, wenn *i* ein Wert ungleich 0 zugewiesen wurde, andernfalls wird *funktion2()* ausgeführt.
Um z.B. auf eine Eingabe für *i* mit dem Wert 32 (Space-Character) durch Aufruf von *funktion2()* zu reagieren, wäre diese Zeile folgendermaßen zu schreiben:

```
(i = getch()) - 32 ? funktion1() : funktion2()
```

Durch das Abziehen des Wertes 32 von der Eingabe für *i* wird *funktion2()* dann ausgeführt, wenn ein Zeichen mit dem Code 32 eingegeben wurde, da dann die Bedingung den Wert 0 (*false*) annimmt. Man beachte die Stelle, an der 32 abgezogen werden müssen (Klammern sind nicht notwendig, da der Term vor dem Fragezeichen einen abgeschlossenen Ausdruck darstellt).

2.5.2 , (Komma-Operator)

Mitunter müssen in einem Ausdruck mehrere Variablen gleichzeitig gesetzt werden. Dies kann z.B. bei der *for*-Funktion auftreten, wenn mehrere Indizes verwendet werden sollen. Statt

```
i = 1;
j = 1;
for (k = 1; k < 10; k++) ...
```

kann man hier kürzer schreiben:

```
for (i = 1, j = 1, k = 1; k < 10; k++) ...
```

Man beachte die Reihenfolge bzw. die Positionen der Kommata und der Semikolons!
Gegebenenfalls kann es sinnvoll sein, eine Wertzuweisung zur besseren Nachvollziehbarkeit in zwei Schritten zu vollziehen, um ihre Bedeutung hervorzuheben. Statt

```
adr = BASE;
adr += 2;
zuweisung (adr, wert);
```

ließe sich diese Anweisung aber auch in einer Zeile notieren, ohne viel an Lesbarkeit zu verlieren:

```
zuweisung ((adr = BASE, adr += 2), wert);
```

In diesem Beispiel wird deutlich, daß das Komma neben der eben beschriebenen Bedeutung als Zuweisungsseparator auch zur Auflistung von Parametern dient, wie man es von Pascal her kennt.

2.5.3 Zuweisungen

Die Sprache C ist vor allem für ihre Flexibilität im Umgang mit Variablen bekannt. Dies gründet sich nicht nur in der Vereinheitlichung der grundlegenden Datentypen über ihr numerisches Format, sondern auch in einer großen Vielfalt von unterschiedlichen Zuweisungsmöglichkeiten.
Wenn in den folgenden Beispielen nur einbuchstabige Variablennamen benutzt werden, dient das lediglich einer verkürzten Darstellungsform - grundsätzlich kann jeder Variablenname auch durch Ausdrücke mit anderen Variablen oder Funktionen ersetzt werden.

Einfache Zuweisung
Bei der einfachen Zuweisung wird einer Variablen der Wert einer Konstanten oder einer anderen Variablen zugewiesen:

```
x = 3;
x = y;
x = BASEADR;
```

Der zuzuweisende Wert darf auch durch Operationen mit Konstanten oder Variablen errechnet werden, die zuzuweisende Variable darf ihrerseits im Argument enthalten sein:

```
x = 2 * x;
x = BASEADR - (OFFSET + 1) * 2;
```

Operative Zuweisung
Diese Art der Zuweisung benutzt eine verkürzte Schreibweise, grundsätzlich läßt sie sich auch durch eine einfache Zuweisung ersetzen. Jede Zuweisung, die einer Variablen einen Wert zuweist, der durch eine einfache Operation aus dem alten Wert hervorgeht, läßt sich folgendermaßen darstellen:

```
x += 2;      /* entspricht: x = x + 2;         */
x *= 3;      /* entspricht: x = x * 3;         */
x %= 3;      /* entspricht: x = x % 3);        */
x *= y + 1;  /* entspricht: x = x * (y + 1);   */
```

Inkrement- und Dekrement-Operatoren
Eine häufig benötigte Zuweisung ist das Erhöhen (Inkrement) oder Erniedrigen (Dekrement) einer numerischen Variablen um 1. Während in Pascal die Befehle *inc(var,[step])* bzw. *dec(var,[step])* vorhanden sind, verzichtet C auf die optionale Wahlmöglichkeit des Wertes für *step*, der bei Pascal auf 1 voreingestellt ist. Für Werte größer als 1 sind in C deshalb die operativen Zuweisungen zu verwenden.

Für den Fall step = 1 benutzt C folgende Notierung:

```
++x;       /* entspricht: x = x + 1;   */
--x;       /* entspricht: x = x - 1;   */
x++;       /* entspricht: x = x + 1;   */
x--;       /* entspricht: x = x - 1;   */
```

Soll durch diese Zeile nur der Wert der Variablen verändert werden, spielt die Positionierung der doppelten Operationszeichen keine Rolle. Sind sie jedoch Bestandteil eines Funktionsargumentes, ergeben sich Unterschiede in der Reihenfolge der Ausführung. Zum Beispiel wird bei der Zeile

```
i = 9;
while (++i < 10).../* Vergleich und Durchlauf mit i=10      */
```

zuerst i um 1 erhöht, *anschließend* der Vergleich mit 10 durchgeführt und somit die folgende Anweisung nicht mehr ausgeführt, während bei

```
i = 9;
while (i++ < 10).../* Vergleich mit i=9, Durchlauf mit i=10*/
```

zuerst der Vergleich von i = 9 mit dem Wert von 10 durchgeführt wird, der die Ausführung der Anweisung signalisiert, *anschließend* wird *i* um 1 erhöht und dann die Anweisung (nun aber mit *i = 10!*) ausgeführt. Dies gilt auch bei der Verwendung dieser Operatoren bei Argumenten von Funktionen, so wird bei

```
i = 0;
ausgeben (i++);          /* Ausgabe von 0, dann i erhöhen    */
ausgeben (++i);          /* i erhöhen, dann Ausgabe von 2    */
```

im ersten Aufruf von *ausgeben()* für *i* ein Wert von 0 ausgegeben (die Erhöhung erfolgt anschließend), während beim zweiten Aufruf ein Wert von 2 ausgegeben wird, da nun erhöht wird, bevor die Ausgabe erfolgt.

Inkrement- und Dekrement-Operatoren bei Zeigern

Handelt es sich bei der zu erhöhenden oder zu erniedrigenden Variablen um einen *Zeiger* auf einen Datentyp, verhalten sich diese Operatoren konsistent. Statt die Adresse um den numerischen Wert 1 zu verändern, wird statt dessen der Zeiger auf das nachfolgende oder vorausgehende Zeigerelement gerichtet. Ist *p* z.B. ein Zeiger auf Variablen vom Typ *long*, so liefert *p*++ den nächsten *long*-Eintrag (eine Validitätsprüfung findet nicht statt!), verändert also den Adresswert von *p* um 4 (das ist die Größe eines *long*-Eintrags in Byte) statt nur um 1.

Rangfolge von Operationen und Operatoren

So wie in der Mathematik die Regel "Punktrechnung vor Strichrechnung" gilt, binden die einzelnen Operationen und Operatoren in C ebenfalls unterschiedlich stark an ihre Operanden. Die Level der einzelnen Operanden sind auf den Seiten 5 und 6 angegeben, je kleiner der Level, desto stärker ist die Bindung. Soll eine andere Bindung erzwungen werden, müssen Klammern gesetzt werden. Nachfolgend einige Beispiele:

```
y = ++x * 2;        /* erst x um 1 erhöhen, dann mit 2 multipl.  */
y = ++(x + 2);      /* erst x mit 2 multipl., dann 1 addieren    */
```

Operationen gleichen Levels werden von rechts nach links abgearbeitet, wie es folgendes Beispiel plausibel macht:

```
x = 4 << 2 << 1;         /* ergibt 64   */
```

Um den Wert für x zu bestimmen, sollen die Bits der Zahl 4 nach links geschoben werden. Rechts vom ersten << Zeichen findet sich die Anzahl der zu verschiebenden Stellen, die sich dadurch ergibt, daß die Bits der Zahl 2 um 1 Stelle nach links geschoben werden sollen. Eine Klammersetzung verdeutlicht dies:

```
x = 4 << (2 << 1);       /* ergibt 64   */
```

Sollte eine andere Reihenfolge verlangt werden, müssen die Klammern anders gesetzt werden:

```
x = (4 << 2) << 1;       /* ergibt 32   */
```

3. Ausgabe- und Eingabe-Befehle

Zur Kommunikation mit dem Benutzer müssen Programme in der Lage sein, Informationen auf einem geeigneten Gerät auszugeben. Je nach Gerätetyp stellt C unterschiedliche Funktionen zur Verfügung, die sich grundsätzlich in 2 Kategorien einteilen lassen: Zeichenorientierte Ausgabe und Blockorientierte Ausgabe. Typische Geräte für zeichenorientierte Ausgaben sind ein Bildschirm oder ein Drucker, während zu den blockorientierten Geräten Festplatten oder Netzwerksysteme zählen.

3.1 Zeichenorientierte Ausgabebefehle

Hiermit sind Ausgabebefehle gemeint, die ein einzelnes Zeichen oder eine Zeichenkette auf das jeweilige Gerät ausgeben. Die Ausgabe einer Zeichenkette entspricht dabei dem sogenannten *streamed output*, was bedeutet, daß die Zeichen, die die Zeichenkette bilden, nacheinander an das Gerät übertragen werden. Unter bestimmten Voraussetzungen kann ein blockorientiertes Gerät (sog. *block device*) sich wie ein zeichenorientiertes Gerät (sog. *char device*) verhalten. Je nach Sprachdialekt gibt es verschiedenste Befehle in C, die die zeichenorientierte Ausgabe unterstützen.

3.1.1 *int* ***putch(****int ch****)***

Mit diesem Befehl wird das im Lowbyte von *ch* enthaltene Zeichen als ASCII-Character direkt auf dem Bildschirm ausgegeben. Trat bei der Ausgabe ein Fehler auf, wird von der Funktion der Wert *EOF* (das ist das Steuerzeichen ^Z bzw. *0x1A*) zurückgegeben, ansonsten der Code des ausgegebenen Zeichens.

3.1.2 *int* ***putchar(****int ch****)***

Diese Funktion arbeitet genauso wie die Funktion *putch()* mit dem Unterschied, daß die Ausgabe auf das Gerät erfolgt, das als *stdout* deklariert ist. Üblicherweise ist das zwar der Bildschirm, jedoch kann durch eine Umbenennung (sog. Ausgabeumleitung von DOS) die Ausgabe auch in eine Datei erfolgen.
Beispiel: Ein Programm wird z.B. mit folgender DOS-Zeile aufgerufen:

```
test > test.err
```

so erscheinen alle Ausgaben, die mit *putch()* gemacht werden, auf dem Bildschirm, jedoch alle mit *putchar()* gemachten Ausgaben werden in die Datei *test.err* geschrieben.

3.1.3 *int* ***putc(****int ch, FILE *stream****)***

Noch mehr Flexibilität bietet diese Funktion, da in ihr jedes beliebige Ausgabegerät gewählt werden kann. Die Angabe eines (notwendigerweise bereits geöffneten) "Datenstromes" für die Ausgabe läßt die Ausgabe auf Bildschirm, Drucker oder Datei zu. So ist zum Beispiel der Funktionsaufruf

```
putc(ch,stdout)
```

gleichbedeutend mit dem Aufruf

```
putchar(ch)
```

3.1.4 *int* ***puts(****char *string****)***

Diese Funktion gibt die in *string* enthaltene Zeichenkette aus, die durch ein 0-Character ('\0') abgeschlossen sein muß. Man beachte, daß *string* hierbei einen Zeiger auf einen Speicherbereich darstellt, der vom Typ *char* gebildet wird. Durch den 0-Character muß der Speicherbereich für eine Zeichenkette um 1 Byte größer sein als die Anzahl der auszugebenden Zeichen der Zeichenkette. Der Rückgabewert der Funktion ist 0, wenn die Ausgabe erfolgreich war, ansonsten ungleich 0. Wird eine konstante Zeichenkette angegeben, wird der 0-Character automatisch angefügt. Wird die Zeichenkette jedoch auf andere Weise erzeugt, muß dieses Zeichen am Ende der Zeichenkette angehängt werden, da sonst eine Ausgabe solange erfolgt, bis ein 0-Character gefunden wird.
Die folgende Anweisung gibt einen konstanten Text aus:

```
puts("Hallo, dies ist ein Test.");
```

*3.1.5 int **printf(**const char *format[, argument]...**)***

Dieser zeichenorientierte Ausgabebefehl ist der leistungsfähigste in seiner Art, den die Sprache C zu bieten hat. Durch die Verwendung eines Formatstrings erlaubt er nicht nur die Ausgabe von Zeichen oder Zeichenketten, sondern auch die formatierte Ausgabe von numerischen Werten, die als Konstanten oder Variable vorliegen können. Die Ausgabe erfolgt auf das als *stdout* definierte Gerät.
Für die Bildung des Formatstrings stehen drei Komponenten zur Verfügung:

1. ASCII-Zeichen zur Bildung eines konstanten Textes

2. Spezielle Steuercodes zur Darstellung von Sonderzeichen sowie zur Positionierung des Textes, diese werden in jedem Fall durch einen *Backslash* \ eingeleitet

3. Spezielle Formatanweisungen zur Ausgabe des Inhaltes von numerischen Variablen und Zeichenketten, diese werden in jedem Fall durch ein *Prozentzeichen %* eingeleitet.

Da die Zeichen \ und % als spezielle Steuerzeichen verwendet werden, erfolgt ihre Ausgabe als Bestandteil eines konstanten Textes etwas umständlicher: damit sie während der Abarbeitung des Formatstrings als ASCII-Codes und nicht als Einleitungszeichen für die Ausgabe eines Steuerzeichens oder einer Variablen interpretiert werden, müssen sie wiederholt werden:

```
printf("Dieser Text gibt ein Prozentzeichen %% aus");
```

bzw.

```
printf("Dieser Text gibt einen Backslash \\ aus");
```

Der übrige Text wird als konstanter Text ausgefaßt und ausgegeben.

Die Einfügung von Steuerzeichen erlaubt es, Tabulatoren zu verwenden, eine neue Zeile zu beginnen oder die Schreibmarke im Text zu bewegen, um den Text zu gliedern. Die Bedeutungen der zulässigen Steuersequenzen sind im Abschnitt "Steuercodes (Escape-Sequenzen)" auf Seite 6 unten zusammengestellt. Sie dürfen beliebig kombiniert werden. Zum Beispiel wird die Anweisung

```
printf("Ausgabe: \nEingabe:");
```

zu folgender Ausgabe führen:

```
Ausgabe:
Eingabe:
```

Man beachte, daß hinter dem Steuercode *\n* kein Leerzeichen steht, denn dieses würde als erstes auszugebendes Zeichen in der zweiten Zeile das Wort *Eingabe* um eine Stelle nach rechts versetzen.

Sind in dem Formatstring Anweisungen zur Ausgabe von Variablen enthalten (erkennbar an einem führenden Prozentzeichen), müssen die Variablen, durch Kommata getrennt, im Anschluß an den Formatstring in der entsprechenden Reihenfolge angegeben werden. Die Funktion *printf()* führt keine Überprüfung durch, ob die angegebenen Variablentypen mit den Formatangaben im Formatstring konsistent sind!
Die Formatangabe für eine Variable wird grundsätzlich durch das Prozentzeichen eingeleitet und durch einen Buchstaben, der den Variablentyp kennzeichnet, abgeschlossen. Zwischen diesen beiden Zeichen können zusätzliche Parameter angegeben werden, die das Ausgabeformat, wie zum Beispiel vorangestellte Vorzeichen, konstante Stellenzahl, links- oder rechtsbündige Ausrichtung, Zahlenformat, Nachkommastellen usw. bestimmen. Somit sind für die Plazierung einer Variablenausgabe im Formatstring mindestens 2 Zeichen notwendig, wie das folgende Beispiel zeigt:

```
i = 13;
printf("Ergebnis: %d",i);
```

man erhält als Ausgabe:

```
Ergebnis: 13_
```

(Der Unterstrich soll angeben, daß sich hinter der 13 die Schreibmarke befindet).

Die etwas unübersichtliche allgemeine Syntax für die Formatangabe von Variablen ist

```
%[-][+|SPACE][#][width][.prec][model][prefix]type
```

hierbei bezeichnet SPACE ein Leerzeichen, während die übrigen Bezeichner durch entsprechende Werte bzw. Abkürzungen zu ersetzen sind. Bei der Eingabe der entsprechenden Anweisungen ist zu beachten, daß (wie sonst auch in C) zwischen Klein- und Großschreibung unterschieden werden muß!
Die Bedeutung der Zeichen im Einzelnen:

- `-` Linksbündige Ausrichtung (nur sinnvoll bei Angabe von *width*)
- `+` positive Zahlen mit einem vorangestellten Pluszeichen versehen
- `SPACE` positive Zahlen mit einem vorangestellten Leerzeichen versehen
- `#` hexadezimale oder oktale Zahlen mit einer führenden 0 ausgeben
- *width* Anzahl der Zeichen für die Ausgabelänge bei rechts- oder linksbündiger Ausrichtung
- *.prec* Anzahl der Nachkommastellen bei reellen Zahlen
- *model* Diese Angabe betrifft das Speichermodell des Compilers, folgende Angaben sind erlaubt:
  - F Kennzeichnung einer FAR-Variablen bei der Compilereinstellung *small model*
  - N Kennzeichnung einer NEAR-Variablen bei den übrigen Compilereinstellungen
- *prefix* ergänzt die folgende numerische Variablenkennung nach folgender Vereinbarung:
  - h Variablen vom Typ *int* werden als *short int* interpretiert
  - l Variablen vom Typ *int* werden als *long int*, vom Typ *float* als *double* interpretiert
  - L Variablen vom Typ *float* werden als *long double* interpretiert
  - N Ausgabe des Offset-Wertes eines *pointers* (nur bei nachfolgendem Typ *p*)
- *type* kennzeichnet den Variablentyp, der für die Ausgabe verwendet werden soll, entsprechend nachfolgender Auflistung:
  - c character, Ausgabe des Variablenwertes als ASCII-Zeichen
  - d dezimal integer, es wird der vorzeichenbehaftete Wert als *int* ausgegeben
  - D long decimal integer, es wird der vorzeichenbehaftete Wert als *long* ausgegeben
  - e signed exponential, Ausgabe einer reellen Zahl im Format *[-]d.dddddd**e**[±]ddd*
  - E signed exponential, Ausgabe einer reellen Zahl im Format *[-]d.dddddd**E**[±]ddd*
  - f signed floating point, Ausgabe einer reellen Zahl im Format *[-]dddd.dddd*
  - g automatic, Exponent zwischen -4 und 4: Ausgabe wie bei f, sonst wie bei e
  - G automatic, Exponent zwischen -4 und 4: Ausgabe wie bei f, sonst wie bei E
  - i dezimal integer, Ausgabe wie bei d
  - o unsigned octal integer, Ausgabe einer *int* Variablen in oktalem Zahlenformat
  - O unsigned octal long, Ausgabe einer *long* Variablen in oktalem Zahlenformat
  - p pointer, Ausgabe der Adresse eines *pointers* im hexadezimalen Format *dddd:dddd*
  - s string, Ausgabe einer mit *\0* abgeschlossenen Zeichenkette
  - u unsigned decimal, dezimale Ausgabe einer vorzeichenlosen *int* Variablen
  - U unsigned long, dezimale Ausgabe einer vorzeichenlosen *long* Variablen
  - x unsigned hexadecimal, hexadezimale Ausg. einer vorzeichenlosen *int* Variablen
  - X unsigned hexadecimal long, hexadez. Ausg. einer vorzeichenl. *long* Variablen

Nachfolgend einige Beispiele und deren Ausgabeformat auf dem Bildschirm:

```
printf("= (%f)"),3.1415926);          = (3.141593)
printf("= (%6.3g)"),3.1415926);       = (  3.14)
printf("= (%+.3E)"),3.1415926);       = (+3.142E+000)

printf("= (%d)"),314);                = (314)
printf("= (%-8d)"),314);              = (314     )
printf("= (%#o)",314);                = (0472)
printf("= (%#X)",314);                = (0x13A)
printf("= (%x)",314);                 = (13a)

printf("= (%c %hd %#hx)",c,c,c);      = (A 65 0x41)
```

3.2 Zeichenorientierte Eingabebefehle

Hiermit sind Eingabebefehle gemeint, die ein einzelnes Zeichen oder eine Zeichenkette von einem entsprechenden Gerät entgegennehmen. Die Eingabe einer Zeichenkette entspricht dabei dem sogenannten *streamed input*, was bedeutet, daß die Zeichen, die die Zeichenkette bilden, nacheinander von dem Gerät übertragen werden. Unter bestimmten Voraussetzungen kann ein blockorientiertes Gerät (sog. *block device*) sich wie ein zeichenorientiertes Gerät (sog. *char device*) verhalten. Je nach Sprachdialekt gibt es verschiedenste Befehle in C, die die zeichenorientierte Ausgabe unterstützen.

*3.2.1 int **getch( );***

Mit diesem Befehl wird ein Zeichen von der Console-Tastatur eingelesen, es erfolgt jedoch keine Ausgabe auf dem Bildschirm (sog. *Eingabe ohne Echo*). Um eine Funktionstaste einlesen zu können, muß diese Funktion zweimal aufgerufen werden, da beim ersten Aufruf das 0-Character und anschließend der Tastencode übertragen werden.

*3.2.2 int **getche( );***

Dieser Befehl hat die gleiche Wirkung wie der Befehl *getch( )*, aber das eingegebene Zeichen wird sofort auf dem Bildschirm ausgegeben. Damit kann der Befehl

```
ch = getche();
```

ersetzt werden durch die Sequenz

```
putch(ch = getch());
```

*3.2.3 int **getchar();***

Mit diesem Befehl wird ein Zeichen von der Eingabeeinheit eingelesen, die durch die vordefinierte Eingabeeinheit *stdin* definiert ist. Üblicherweise ist das zwar die Tastatur, jedoch kann durch eine Umbenennung (sog. Eingabeumleitung von DOS) die Eingabe auch aus einer Datei erfolgen.
Beispiel: Wird ein Programm z.B. mit folgender DOS-Zeile aufgerufen:

```
test < test.inp
```

so werden alle Eingaben mittels *getchar()* sequentiell aus der Datei mit dem Namen *test.inp* gelesen, während mit *getch()* nach wie vor von der Console-Tastatur gelesen wird.

*3.2.4 int **getc(**FILE *stream**);***

Noch mehr Flexibilität bietet diese Funktion, da in ihr jedes beliebige Eingabegerät gewählt werden kann. Die Angabe eines (notwendigerweise bereits geöffneten) "Datenstromes" für die Eingabe läßt Eingaben von Tastatur, Terminal oder Datei zu. Demnach ist zum Beispiel der Funktionsaufruf

```
ch = getchar()
```

gleichbedeutend mit dem Aufruf

```
ch = getc(stdin)
```

*3.2.5 char ***gets(**char *buffer**);***

Diese Funktion liest von der Standard-Eingabeeinheit *stdin* eine Zeichenfolge in den Zwischenspeicher *buffer* ein, bis die Eingabe mit <RET> (*\n*) abgeschlossen wird. Das Zeichen *\n* wird durch ein 0-Character ersetzt und ein Zeiger auf den Eingabepuffer *buffer* zurückgegeben. Bei einem Fehler während der Eingabe der Zeichenkette wird statt des Argumentes (das ist der Zeiger auf *buffer*) ein *NUL*-Zeiger zurückgegeben. Im folgenden Beispiel wird nach einer Eingabeaufforderung eine Zeichenkette eingelesen und anschließend wieder ausgegeben:

```
main()
  { charbuffer[100];
    char *kette;
    printf("Zeichenkette eingeben: ");
    kette = gets(buffer);
    printf("\nEingegeben wurde: %s",kette);
  }
```

*3.2.6 int **scanf(**const char *format [,argument]...**)***

Wie die bereits beschriebene Funktion *printf()* stellt die Funktion *scanf()* ein leistungsfähiges Kommando zur Eingabe von Werten von der als *stdin* definierten Eingabeeinheit dar. Die Daten werden aus dem eingehenden Datenstrom sequentiell an die Speicherstellen geschreiben, die durch die angegebenen Argumente spezifiziert sind. Dabei muß jedes Argument einen Zeiger auf eine Variable darstellen, deren Typ mit dem im Formatstring definierten Typ vereinbar ist.

Für die Bildung des Formatstrings gelten folgende Regeln:

1. Die Angabe eines Leerzeichens bewirkt, daß bei der Eingabe auftretende Leerzeichen nicht für die Zuweisung zu einer Variablen mitgelesen werden. Erlaubt, aber ohne Auswirkungen auf die Eingabe oder die Zuweisung sind die Steuercodes *\t* (Tabulator) und *\n* (Return), die ebenfalls zur Trennung mehrerer Eingaben zugelassen sind (im Weiteren werden diese Zeichen ebenfalls als Leerzeichen bezeichnet).

2. Das Prozentzeichen *%* identifiziert im Formatstring eine Variable und die zugehörige Variablendeklaration, die mit dem Typ des entsprechenden Argumentes übereinstimmen muß. Die möglichen Parameter ähneln denen von *printf()* und werden weiter unten erläutert. Die Eingabe einer Variablen endet mit dem ersten Leerzeichen, durch Eingabe eines Zeichens, das nicht zum Typ der Variablen paßt, oder durch Erreichen der maximalen Eingabelänge, soweit definiert.

3. Alle anderen ASCII-Zeichen im Formatstring müssen bei der Eingabe entsprechend ihrer Reihenfolge und Position im Formatstring eingegeben werden, ansonsten werden keine weiteren Zuweisungen zu den Argumenten vorgenommen - eine Anzeige, um welche Zeichen es sich handelt, erfolgt nicht! Die Eingabe endet in diesem Fall nur dann nach der ersten fehlerhaften Eingabe, wenn anschließend die Taste <RET> gedrückt wird.

Die Deklaration einer Variablen mittels des Prozentzeichens erfolgt nach folgender Syntax:

```
%[*][width][model][prefix]type
```

Die auf das Prozentzeichen folgenden Parameter in eckigen Klammern sind wahlfrei und haben folgende Bedeutung:

*	Das folgende Eingabefeld wird bei der Zuweisung von Variablen ignoriert
width	Anzahl der maximal einzugebenden Zeichen. Es werden weniger Zeichen eingelesen, wenn bis dahin ein Leerzeichen oder ein nicht zum Variablentyp passendes Zeichen eingegeben wird.
model	Diese Angabe betrifft das Speichermodell des Compilers, folgende Angaben sind erlaubt: F Kennzeichnung einer FAR-Variablen bei der Compilereinstellung *small model* N Kennzeichnung einer NEAR-Variablen bei den übrigen Compilereinstellungen
prefix	ergänzt die folgende numerische Variablenkennung nach folgender Vereinbarung: h Variablen vom Typ *int* werden als *short int* interpretiert l Variablen vom Typ *int* werden als *long int*, vom Typ *float* als *double* interpretiert
type	kennzeichnet den Variablentyp für die gewünschte Eingabe, dadurch wird auch der für die Umwandlung zulässige Zeichensatz bestimmt. In der umseitigen Tabelle sind die erwarteten Eingaben bzw. die zulässigen Zeichen sowie den notwendigen Datentyp für das Argument zusammengestellt.

type	Datentyp	Erwartete Eingabe
c	pointer to char	Alle ASCII-Zeichen (inclusive Leerzeichen!) (Um ein Zeichen zu lesen, das kein Leerzeichen ist, den Typ *%1s* benutzen)
d	pointer to int	Ziffern 0 ... 9, -
D	pointer to long	Ziffern 0 ... 9, -
o	pointer to int	Ziffern 0 ... 7
O	pointer to long	Ziffern 0 ... 7
x	pointer to int	Ziffern 0 ... 9, Buchstaben a ... f bzw. A ... F (Eine hexadezimale Zahl muß ohne führendes *0x* bzw. *0X* eingegeben werden)
X	pointer to int	Ziffern 0 ... 9, Buchstaben a ... f bzw. A ... F (Eine hexadezimale Zahl muß ohne führendes *0x* bzw. *0X* eingegeben werden)
i	pointer to int	Die Zahlenbasis wird automatisch nach folgendem Schema erkannt: führende 0 für Oktalzahl (0 ... 7) erste Ziffer ungleich 0 ergibt Dezimalzahl (0 ... 9, -) führendes 0x oder 0X ergibt Hexadezimalzahl (0 ... 9, A ... F, X)
I	pointer to long	Die Zahlenbasis wird automatisch nach folgendem Schema erkannt: führende 0 für Oktalzahl (0 ... 7) erste Ziffer ungleich 0 ergibt Dezimalzahl (0 ... 9, -) führendes 0x oder 0X ergibt Hexadezimalzahl (0 ... 9, A ... F, X)
u	pointer to unsigned int	Ziffern 0 ... 9
U	pointer to unsigned long	Ziffern 0 ... 9
e, E *f* *g, G*	pointer to float	Unhabhängig von dem gewählten Typ werden folgende Eingabeformate automatisch richtig erkannt: [-]ddd.ddd Ziffern 0 ... 9, Dezimalpunkt, Vorzeichen [-]ddd.ddd**e**[±]ddd Ziffern 0 ... 9, e, Dezimalpunkt, Vorzeichen [-]d.ddd**E**[±]ddd Ziffern 0 ... 9, E, Dezimalpunkt, Vorzeichen Die Eingabe des Pluszeichens vor dem Exponenten ist wahlfrei.
s	pointer to char[]	Alle ASCII-Zeichen (jedoch keine Leerzeichen!), der 0-Character (*end of string, EOS*) wird automatisch angehängt.
p	pointer to far pointer	Eingabe einer hexadezimalen Zahl nach dem Format xxxx:yyyy, zulässig: Ziffern 0 ... 9, Buchstaben A ... F (Großbuchstaben!)
n	pointer to int	Es werden keine Zeichen der Eingabe gelesen, jedoch wird in der entsprechenden Variablen die Zahl der bislang erfolgreich gelesenen Zeichen abgelegt

Wird bei den numerischen Typen ein *prefix* (*h* oder *l*) verwendet, so werden die jeweiligen Datentypen entsprechend als *Short*- oder *Long*-Version behandelt.

Für die Behandlung von String-Variablen gelten folgende weiteren Zusätze:

*%**[set of char]**s*	Durch Angabe einer Menge von Zeichen in eckigen Klammern wird festgelegt, welche Zeichen für die Eingabe erlaubt sind. Diese Menge darf auch das Leerzeichen enthalten, welches sonst die Eingabe beendet.
*%**[^set of char]**s*	Das vorangestellte Caree-Zeichen ^ kehrt die Bedeutung der Menge um: Wird eines dieser Zeichen eingegeben, wird die Eingabe beendet.
*%**n**c*	Hiermit wird einer String-Variablen eine Zeichenkette zugewiesen, ohne daß ein 0-Character angefügt wird. Die zugeordnete Variable muß jedoch vom gleichen Typ sein wie bei *%s*.

Im Folgenden sind einige Beispiele für die Anwendung von *scanf()* mit den entsprechenden Eingaben und den sich ergebenden Zuweisungen aufgeführt. Für alle Beispiele gelten folgende Deklarationen:

```
int x,y
char ch;
char strg[20];
```

In den Beispielen wurde das abschließende <RET> bei den Eingaben weggelassen, ansonsten sind exakt die Eingaben (inclusive Leerzeichen) aufgeführt, die zu den unter Ergebnis genannten Werten für die Variablen führen.

Befehlszeile:	Eingabe:	Ergebnis:
`scanf("%d",&x);`	*12*	*x = 12*
`scanf("%d %d",&x,&y);`	*12 13*	*x = 12, y = 13*
`scanf("%d %d",&x,&y);`	*12, 13*	*x = 12, y = 0*
`scanf("%i",&y);`	012	x = 10
`scanf("%i",&y);`	0x12	x = 18
`scanf("%i",&y);`	12	x = 12
`scanf("%x:%x",&x,&y);`	0xC000:0x1234	x = 0xC000, y = 0x1234
`scanf("%x:%x",&x,&y);`	0xC000 0x1234	x = 0xC000, y = 0x0000
`scanf("%s",strg);`	ABCDEF	strg = "ABCDEF"
`scanf("%4s",strg);`	ABCDEF	strg = "ABCD"
`scanf("%3c",strg);`	ABCDEF	strg = "ABC"
`scanf("%[abc]s",strg);`	abcdef	strg = "abc"
`scanf("%[^e]s",strg);`	abcdef	strg = "abcdf"
`scanf("%d%s",&x,strg);`	123 123	x = 123, strg = "123"
`scanf("%d%s",&x,strg);`	123 ZYX	x = 123, strg = "ZYX"
`scanf("%d%s",&x,strg);`	0x123	x = 0, strg = "x123"
`scanf("%i%s",&x,strg);`	0x123 123	x = 291, strg = "123"

Man beachte, daß bei der Verwendung als Argument für *scanf()* die Variablen *x, y* und *ch* mit einem Und-Zeichen *&* versehen sein müssen, damit deren assoziierte Zeiger übergeben werden, während bei der Variablen *strg* dies nicht nötig ist, da sie bereits *per definitionem* als Zeiger deklariert ist, erkenntlich an den eckigen Klammern bei der Definition von *strg*.

Die Funktion *scanf()* gibt als Rückegabewert die Anzahl der Felder zurück, die erfolgreich umgewandelt wurden. Dies kann man ausnutzen, um Eingabefehler zu erkennen:

```
if (scanf("%d %s",&y,strg) < 2) printf("Eingabefehler!");
```

Wird z.B. die Zeichenfolge

```
C800 Test          ergibt: x = 1, y = (undefiniert), strg = "C800"
```

eingegeben, erfolgt keine Zuweisung der ersten Eingabe zu *y*, da eine Dezimalzahl erwartet wurde, der erste Buchstabe aber keine Ziffer ist. Statt dessen wird die Eingabe *C800* als Text der nächsten Variablen, nämlich *strg* zugeordnet und der Rest der Eingabe (*Test*) für den nächsten Aufruf von *scanf()* zurückgestellt. Der Rückgabewert von *scanf()* ist jedoch 1, während er bei einer richtigen Eingabe den Wert 2 gehabt hätte:

```
1234 Test          ergibt: x = 2, y = 1234, strg = "Test"
```

liefert dabei ebenso ein korrektes Resultat wie die Eingabe

```
1234Test           ergibt: x = 2, y = 1234, strg = "Test"
```

4. Die Arbeit mit Syntax-Graphen

Eine Programmiersprache ist eine formalisierte Sprache, die mit den darstellerischen Mitteln allgemein verständlicher Symbolik (wie z.B. Buchstaben und Zeichen) gebildet wird. Dabei ist es unabdingbar notwendig, eine eindeutige Zuordnung zwischen Darstellung und Begriff zu schaffen - eine Mehrdeutigkeit von Begriffen ist in einer Programmiersprache nicht zulässig, eine kontextabhängige Sinnverschiebung, wie sie in gesprochenen Sprachen die Regel ist, darf nicht auftreten.
Syntax-Graphen geben in einer symbolischen Form an, nach welchen Gesetzmäßigkeiten Begriffe definiert und durch eine vereinbarte Menge von Symbolen dargestellt werden dürfen. Hierbei ergeben sich zwei Eckpunkte:

- die kleinste Symboleinheit des Darstellungsmediums (z.B. ein Buchstabe)
- der allgemeinste Begriff für ein Element der Programmiersprache (z.B. ein Programm)

Aus den oben genannten Bedingungen ergibt sich, daß ein diesen Regeln entsprechendes Regelwerk den Charakter eines Algorithmus haben muß. Es ist deshalb ein wesentlicher Bestandteil eines jeglichen Compilers, eine syntaktische Prüpfung durchzuführen, ob der eingegebene Programmcode diesem Regelwerk entspricht oder nicht. In einen ausführbaren Programmcode läßt sich ein Programmtext nur dann überführen, wenn die angebenenen Statements den syntaktischen Regeln der Sprache entsprechen. Insgesamt definieren Syntax-Graphen dabei nicht die Sprache*per se*, sondern stellen lediglich die Regeln dar, nach denen Elemente dieser Sprache konstruiert werden dürfen.

4.1 Bedeutung der Symbolik

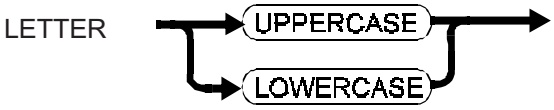
Ein Syntax-Graph besteht aus dem Bezeichner, der durch den Graphen definiert werden soll, mindestens einem weiteren bereits definierten Bezeichner und einem System von Pfeilen, das eine Gesetzmäßigkeit für die Anordnung bereits definierter Bezeichner angibt, um den gewünschten Begriff zu spezifizieren. Die einfachste Anordnung ergibt sich z.B. durch

womit ein einzelnes Zeichen (hier:_) definiert wird. Der von links kommende Pfeil beschreibt dabei den Weg, über den der zu definierende Begriff dargestellt werden soll, in dem Kreis (oder Oval) steht ein allgemein als eindeutig anerkannter Grundbegriff oder ein anderer bereits definierter Begriff, der nach rechts weisende Pfeil zeigt an, daß damit die Definition des neuen Begriffes abgeschlossen ist.
In vielen Fällen sind mehrere Möglichkeiten vorhanden, mit denen ein neuer Begriff inhaltlich festgelegt werden kann, z.B.:

das heißt, ein Kleinbuchstabe (LOWERCASE) wird durch genau einen der Buchstaben a, b, c, ... z gebildet, ein Großbuchstabe (UPPERCASE) durch genau einen der Buchstaben A, B, C, ..., Z.

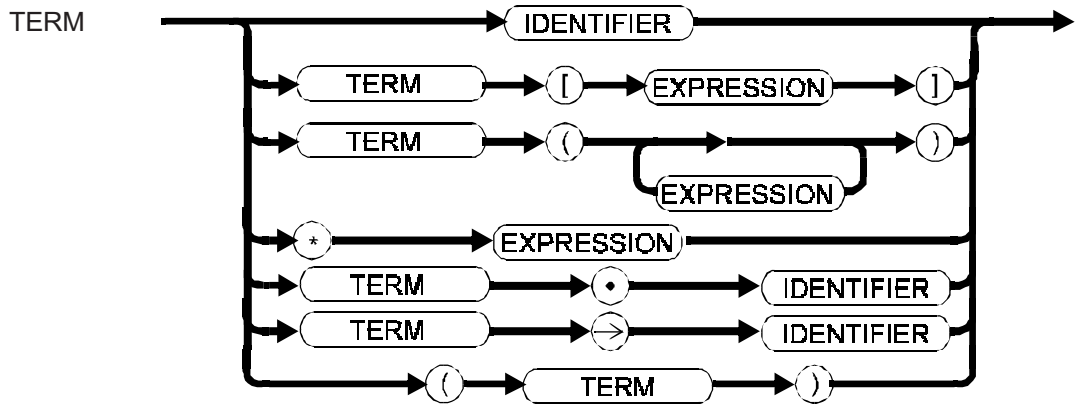
4.2 Benutzung von Syntax-Graphen

Das Bildungsprinzip für Syntax-Graphen kann nun an dem Begriff des Buchstabens deutlich gemacht werden:



Demnach besteht ein Buchstabe (LETTER) entweder aus einem Kleinbuchstaben (LOWERCASE) oder einem Großbuchstaben (UPPERCASE), wie sie zuvor bereits definiert worden sind.

Ein wesentliches Kriterium flexibler Sprachen ist die Möglichkeit der rekursiven Sprachbeschreibung, das heißt, die Definition eines neuen Begriffes darf den zu definierenden Begriff enthalten, sofern er aus mindestens einem weiteren, bereits definierten Begriff wahlweise gebildet werden kann.
Ein typisches Beispiel ist dabei die Definition des nachfolgenden Begriffes, mit dem beliebige Ausdrücke erzeugt werden können:



Man beachte, daß der Begriff TERM in dieser Definition mehrmals auftritt, daß es aber auch mindestens zwei Alternativen gibt, den Term zu bilden, ohne daß auf einen bereits vorhandenen Term Bezug genommen werden muß (IDENTIFIER und *EXPRESSION).

Der einfachste Term ist hier ein einfacher Bezeichner, z.B.

```
ZAHL1
```

Ein Term kann aber nach dem Syntax-Graphen auch gebildet werden durch den Ausdruck

```
ZAHL1[ZAHL2]
```

womit man z.B. eine Komponente eines Arrays erhält.
Benutzt man diese Konstruktion wiederum zur Konstruktion eines weiteren Terms, so erhält man z.B.

```
ZAHL3->ZAHL1[ZAHL2]
```

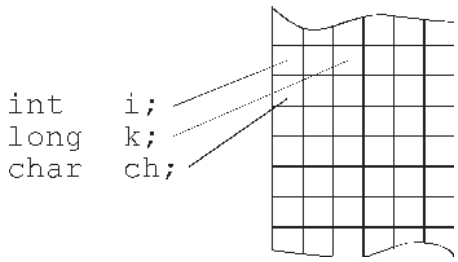
was ZAHL3 als eine Array-Komponente definiert, die ein Bestandteil eines Records ist.

5. Variablen

Zur Speicherung von Informationen muß von einem Programm Speicherplatz zur Verfügung gestellt werden, in dem diese Daten abgelegt werden können. Dabei muß unterschieden werden zwischen *temporärer* und *permanenter* Sicherung der Daten. Für eine permanente Datenspeicherung werden in der Regel Massenspeicher (Diskette, Festplatten, CD-ROM) benutzt, während temporäre Daten im RAM des Rechners abgelegt werden. Erstere bleiben auch nach Beenden des Programmes oder nach Ausschalten des Rechners erhalten, während letztere in diesen Fällen verloren gehen.
Die kleinste Informationseinheit ist das *Bit*, das die Zustände *wahr* oder *falsch* (0 oder 1) annehmen kann. Je acht Bit werden zu einer Organisationseinheit zusammengefaßt, die man als *Byte* bezeichnet. Aus Gründen der Speicheradressierung ist das Byte die kleinste Einheit, auf die ein Prozessor im Speicher zugreifen kann.
Alle in einem Programm auftretenden Informationen müssen in einem Format dargestellt und gespeichert werden, das sich durch ein oder mehrere Byte repräsentieren läßt. Bestimmte, häufig verwendete Formate sind dabei als *elementare Datentypen* vordefiniert und können vom Programmierer unmittelbar verwendet werden. Andere, gegebenenfalls komplexere Datentypen müssen dagegen explizit definiert werden, wobei auf die vorgenannten elementaren Datentypen zurückgegriffen wird.

5.1 Unmittelbar zugängliche Variablen

Die einfachste Deklaration einer Variablen ergibt sich bei den elementaren Datentypen wie z.B.*char*, *int*, *long* usw. Trifft der Compiler auf eine derartige Anweisung, reserviert er einen ausreichend großen Speicherplatz im Datensegment und ordnet den verwendeten symbolischen Namen der zugehörigen Speicherplatzadresse zu, wie dies im nachfolgenden Beispiel veranschaulicht werden soll. In Form einer Tabelle führt der Compiler Buch über jede verwendete Variable und ihren Speicherort. Beim Linken (vergl. S. 4) der Module wird ihrer Umwandlung in ausführbaren Code jeder dieser symbolischen Verweise durch die entsprechende Adresse ersetzt.



symbolischer Name	Adresse	Anzahl Byte
i	DS:0000	2
k	DS:0002	4
ch	DS:0006	1

(DS: Datensegment)

Wird einer dieser Variablen ein Wert zugewiesen, ändert sich unmittelbar der Inhalt der Speicherstelle an der angegebenen Adresse. Nach den Anweisungen

```
i = 300;
k = 0x12345678;
ch = 'A';
```

findet man also an der Speicherstelle DS:0000 den Eintrag 2C 01, der die hexadezimale Darstellung der Zahl 300 darstellt (300d = 012Ch) - hierbei ist zu beachten, daß nach der sog. Intel-Konvention im Speicher zuerst das niederwertige (Low-) und dann das höherwertige (High-) Byte abgelegt wird. An der nächsten Speicherstelle DS:0002 ergibt sich der Eintrag für k - hierbei wird ein long-Wert in zwei Worten dargestellt, wobei das niederwertige Wort zuerst und darin das niederwertige Byte am Anfang abgelegt wird - im Speicher erhält man die Bytefolge 78 56 34 12. Die Zuweisung des Buchstaben 'A' zu ch füllt das Byte an der Adresse DS:0006 mit dem hexadezimalen Wert 41. Insgesamt ergibt sich somit ab der Adresse DS:0000 folgende Bytefolge:

2C	01	78	56	34	12	41
LB	HB	LB	HB	LB	HB	
		LW		HW		

Da die Größe des Datensegmentes auf 64 KByte beschränkt ist, ergibt sich eine obere Schranke für die Menge an Informationen, die während der Laufzeit eines Programmes temporär im Speicher gehalten werden kann, sofern keine weiteren Maßnahmen ergriffen werden.

5.2 Adressen von Variablen

Im vorigen Kapitel wurde bereits deutlich, daß jede Deklaration einer Variablen zusammen mit dem symbolischen Namen eine eindeutige Speicheradresse festlegt, die einen dem Variablentyp angemessenen Speicherbereich bezeichnet. Soll eine Funktion den Wert einer Variablen ausgeben, so genügt es, ihren aktuellen Wert aus der Speicherstelle zu entnehmen und als Parameter der Funktion zu übergeben (sog. call by value). Tatsächlich wird dazu beim Aufruf der Funktion eine neue Variable deklariert (d.h., sie bekommt einen symbolischen Namen und eine zugehörige Speicheradresse) und dieser der Wert zugewiesen, den die zu übergebende Variable hat. Nach Abschluß der Funktion wird dieser Speicherbereich wieder freigegeben. Nimmt die Funktion während ihrer Laufzeit Änderungen am Wert der übergebenden Variablen vor, hat dies lediglich Auswirkungen im Innern der Funktion, nach Beenden der Funktion ist jedoch der ursprüngliche Wert der Variablen unverändert.
In bestimmten Fällen ist es jedoch notwendig, den Wert einer bereits deklarierten Variablen zu ändern oder zuzuweisen. Dazu darf keine Kopie dieser Variablen angefertigt werden, sondern es muß die

Variable selbst übergeben werden - genauer: die Funktion benötigt die Adresse, an der der neue Wert eingetragen werden soll. Zu diesem Zweck existiert in C der sogenannte *Adressoperator* & (genannt ampers-and). Wird einem Variablennamen dieses Zeichen vorangestellt, so wird nicht der Wert der Variablen, sondern die Adresse der Variablen verwendet. Beim Aufruf

```
printf("%d",ch);
```

wird z.B. nach der Deklaration und Zuweisung entsprechend der vorherigen Seite die Zahl 65 ausgegeben, während bei einem Aufruf wie

```
printf("%d",&ch)
```

die *Adresse* der Variablen *ch* (hier das Low-Word), also 6 ausgegeben wird.
Während eine Funktion, die den Wert einer Variablen ausgeben soll, lediglich den Wert und nicht die Adresse einer Variablen benötigt, ist für die Zuweisung, wie sie z.B. bei der Funktion*scanf()* geschieht, die Angabe der Adresse von entscheidender Bedeutung, da ansonsten nach Verlassen der Funktion *scanf()* der ursprüngliche Wert der Variablen nocht erhalten geblieben wäre. Folglich wird bei

```
scanf("%d",&i);
```

der eingegebene Wert richtig an dem Speicherplatz abgelegt, der den Wert der Variablen i bezeichnet, während bei

```
scanf("%d",i);
```

die Eingabe an eine Speicherstelle geschrieben wird, die durch den aktuellen Wert von*i* bestimmt wird. Letzteres führt zu einem unberechenbaren Verhalten des Programmes, falls dabei Teile des Programmcodes überschrieben werden.

5.3 Zeiger auf Variablen

Der Adressoperator kann dazu verwendet werden, einer geeignet deklarierten Variablen die Adresse einer anderen Variablen zuzuweisen. Somit hat die Anweisungsfolge

```
px = &i;
printf("%X",px);
```

die gleiche Wirkung wie die Anweisung

```
printf("%X",&i);
```

denn in beiden Fällen wird die Adresse der Variablen i ausgegeben. Durch die Zuweisung `px = &i` hat *px* den Charakter eines Zeigers auf eine Variable, denn *px* enthält als Wert die Adresse auf einen Speicherplatz, der einen Wert aufnehmen kann. Diesen Zeiger kann man nun benutzen, um der Variablen, auf die *px* zeigt, einen Wert zuzuweisen. Dazu muß bei der Zuweisung aber deutlich gemacht werden, daß man nicht px selbst, sonder der Speicherstelle, auf die px zeigt, einen wert zuweisen will. Der zugehörige Operator ist das * (Stern). Somit ist die Anweisungsfolge

```
 px = &i;
*px =  5;
```

gleichbedeutend mit der Anweisung

```
i = 5;
```

denn in beiden Fällen wird an der Speicherstelle, die mit der Variablen i assoziert ist, der Wert 5 abgelegt.

Damit eine Variable einen Zeiger auf eine Speicherstelle darstellen kann, muß sie als solches deklariert werden. Um sinnvolle Operationen mit diesen Adressen zu ermöglichen, ist es außerdem nötig, anzugeben, auf welchen Typ von Variablen der Zeiger zeigt. Die oben verwendete Zeigervariable*px* zeigt auf eine Variable vom Typ *int*, folglich muß sie deklariert werden als

```
int *px;
```

gelesen "integer pointer px". Die Anweisung px = &i bedeutet "px enthält die Adresse von i" und die Anweisung *px = 5 bedeutet "die Stelle, auf die px zeigt, erhält den Wert 5".
Man beachte, daß jede als Zeiger deklarierte Variable als Adresse auf einen Speicherplatz im Speicher einen Platz von 4 Byte belegt (Segment:Offset), daß aber die Speicherstelle, auf die dieser Zeiger zeigt, erst durch eine Zuweisung während der Laufzeit des Programmes definiert wird.

5.4 Vektoren

Eine besondere Art von Zeigern sind die sogenannten Vektoren. Darunter versteht man eine Anordnung von gleichartigen Variablen unter einem gemeinsamen Namen sowie eine Indizierung, die ein- oder mehrdimensional sein kann. In Pascal bezeichnet man diese Variablen als Arrays.
Ein Vektor ist erkennbar an der Verwendung eckiger Klammern während der Deklaration, dabei wird entweder angegeben, wieviele Elemente der Vektor enthalten soll oder es ergibt sich aus einer Liste von Vordefinitionen, welcher Speicherplatz benötigt wird.
Ist eine Variable als Vektor definiert, so hat die Variable den Typ eines Zeigers, was bedeutet, daß sie an Funktionen grundsätzlich als *call by reference* übergeben wird - das Voranstellen des Adressoperators, z.B. bei *scanf()* ist folglich nicht erlaubt!
Die Deklaration eines Vektors statt eines Zeigers hat den Vorteil, daß bereits bei der Compilierung des Programmes der Speicherplatz festgelegt wird, dabei ergibt sich als Nachteil, daß dieser Speicherplatz auch dann benötigt wird, auch wenn die Variable keine Einträge erhält.
Das folgende Beispiel zeigt die verschiedenen Schritte zur Deklaration und Zuweisung eines Vektors:

```
char strg[10] = "Hallo";
funktionsname()
  {
    printf("%s\n",strg);
    putch(strg[1]);
    scanf("%s",strg);
  }
```

In der ersten Zeile wird *strg* als Vektor auf 10 aufeinanderfolgende Speicherstellen von der Größe eines Byte (char) definiert und damit der zugehörige Speicherort festgelegt. Gleichzeitig wird der Inhalt der ersten 6 Speicherstellen mit den 5 Buchstaben des Wortes "Hallo" sowie einem zusätzlichen, die Zeichenkette abschließenden 0-Character belegt.
In der vierten Zeile wird der Inhalt des Vektors ausgegeben, auf dem Bildschirm erscheint das Wort *Hallo*, gefolgt von einem *carriage return*.
In der fünften Zeile wird mit *putch()* ein einzelnes Zeichen des Vektors ausgegeben. Da *strg* als Vektor von Characters definiert ist, ist eine Komponente des Vektors ein Zeichen, das von einer Funktion wie *putch()* ausgegeben werden kann. Auf dem Bildschirm erscheint der Buchstabe *'a'*.
In der sechsten Zeile wird dem Vektor eine neue Zeichenfolge zugewiesen. Man beachte, daß hier das sonst bei Variablen benötigte &-Zeichen entfällt, da es sich bei *strg* ohnehin um einen Zeiger handelt.
Anmerkung: Die Zuweisung (predefinition) eines Vektors in der hier beschriebenen Weise ist nur außerhalb einer Funktion möglich.

5.5 Zuweisung von Zeigern

Während für Vektoren bereits bei der Übersetzung ausreichender Speicherplatz zur Verfügung gestellt wird, gibt es für Zeiger zwei Möglichkeiten der Zuweisung:

- Während des Programmablaufes wird dem Zeiger als Inhalt der Speicherplatz einer bereits definierten Variablen zugewiesen
- Bei Bedarf wird durch Aufruf einer geeigneten Funktion ein entsprechender Speicherplatz gesucht und dem Zeiger als Inhalt zugewiesen

Die erste Option wird häufig verwendet, um Speicherbereiche unabhängig von ihrem Inhalt manipulieren zu können, während die zweite Option vorwiegend dann eingesetzt wird, wenn der benötigte Speicherplatz für Variable a priori nicht bekannt ist und erst während der Programmlaufzeit ermittelt werden kann.
Die erste Alternative ist bereits im Abschnitt 5.3 vorgestellt worden. Für die zweite Alternative benötigt man eine Funktion, die in der Lage ist, einer als Zeiger deklarierten Variablen eine Adresse zuzuweisen, die auf einen Speicherbereich zeigt, der groß genug ist für die zu speichernde Information. Allgemein bezeichnet man diesen Vorgang als *memory allocation* (Speicherzuweisung), weshalb die entsprechenden Funktionsnamen den Begriff *alloc* in der einen oder anderen Weise enthalten.
Man beachte, daß ein Zeiger ohne vorherige Zuweisung undefiniert ist!

Die meist verwendete Funktion zur Speicherzuweisung ist

```
void *malloc(int size)
```

sie liefert einen Zeiger auf einen Speicherbereich von *size* Byte Größe. Steht nicht genügend Speicherplatz zur Verfügung oder werden mehr als 64 KB angefordert, gibt *malloc()* den Wert *NULL* zurück, das ist ein Zeiger auf die (unzulässige) Adresse 0000:0000.
Der dieser Funktion vorangestellte Datentyp *void* gibt an, daß *malloc()* keinen bestimmten Datentyp zugrundeliegt, sonder durch eine sogenannte *Cast*-Operation dem übermittelten Zeiger nachträglich ein Datentyp zugeordnet werden muß:

```
int *px;
px = (int*) malloc(20);
```

weist der als Zeigervariablen für einen Integer-Bereich deklarierten Variablen px einen Speicherplatz von 20 Byte zu, das heißt, daß darin maximal 10 Integer-Werte Platz finden können.

Wird der durch *malloc()* bereitgestellte Speicherplatz nicht mehr benötigt, kann dieser durch die Funktion

```
void free(void* buffer)
```

wieder freigegeben werden. Im obigen Beispiel für *px* wurden 20 Byte reserviert. Dieser Speicherplatz wird für andere Variablen wieder freigegeben durch den Aufruf

```
free (px);
```

Dabei muß sichergestellt sein, daß der mit *px* übergebene Zeiger tatsächlich definiert ist, da ansonsten unvorhersehbare Speichermanipulationen auftreten können.

5.6 Arithmetik mit Zeigern

In C kann konsequenterweise mit Zeigern genauso gerechnet werden wie mit numerischen Variablen, dabei sind jedoch einige Unterschiede zu beachten, die einerseits ungewohnt, andererseits aber für den Umgang mit Zeigern sehr praktisch sind.
Im weiteren Verlauf gelten folgende Deklarationen:

```
int i,x;        /* einfache numerische Variable */
int a[10];      /* Vektor auf 10 Integerwerte   */
int *pa;        /* Zeiger auf Integerwerte      */
```

Dem Zeiger *pa* wird im Programm die Adresse der Vektors *a* auf die Speicherstelle *a[0]* zugewiesen mit

```
pa = &a[0];
```

Den gleichen Effekt hätte auch die Zuweisung

```
pa = a;
```

denn hier würde *pa* die Adresse des ersten Elementes von *a[]* zugeordnet werden, was nach der Definition eines Vektors aber gleichbedeutend mit dem Zeiger auf die Variable *a* ist.
In der Variablendeklaration ist pa als ein Zeiger auf int-Variablen definiert, die jede einen Speicherplatz von 2 Byte benötigt. Jede zulässige numerische Operation mit Zeigern berücksichtigt diese Tatsache in folgender Hinsicht: Erhöht man den Zeiger um eine Einheit, so erwartet man einen Zeiger auf das nächste Objekt, so wie man bei einem Vektor durch Erhöhen des Index ebenfalls das nächste Element bekommt. Folglich sind folgende Anweisungen äquivalent:

```
x = a[1];
x = *(pa+1)
```

oder, für beliebige Indizes

```
x = a[i];
x = *(pa+i);
```

Intern wird auf die durch die Zeigervariable repräsentierte Adresse bei dieser Operation nicht der Zahlenwert *i*, sondern der Zahlenwert $2 \cdot i$ addiert, da dies der durch die Deklaration definierte Abstand zweier aufeinanderfolgender Einträge ist.

Da es sich bei *a* um einen Vektor (mit festen Adresswert) handelt, sind folgende Ausdrücke unzulässig

```
a = pa;    /* Vektor auf a wird nur durch Deklaration definiert  */
pa = &a;   /* Es gibt keine Adresse von a, nur die von z.B. a[0] */
a = a+1;   /* a ist eine Konstante, keine Veränderung erlaubt    */
```

Das folgende Programmbeispiel zur Bestimmung der Länge einer Zeichenkette soll zeigen, wie die Zeigerarithmetik dazu eingesetzt werden kann:

```
int strlen(s)  /* liefert Anzahl von Zeichen in Zeichenkette s */
char *s;
{
  int n;
  for (n = 0; *S != '\0'; s++) n++;
  return(n);
}
```

Die zu untersuchende Zeichenkette wird in Form eines Zeigers an die Funktion übergeben, wobei *s* auf den Anfang der Zeichenkette zeigt. Beginnend bei 0 wird das entsprechende Zeichen getestet, ob es sich um den 0-Character handelt, der das Ende der Zeichenkette markiert. Ist dies nicht der Fall, wird *n* um 1 erhöht und der Zeiger *s* zeigt durch Verwendung des ++-Operators auf das nächste Zeichen, andernfalls wird die Schleife verlassen und der aktuelle Wert von *n* zurückgegeben.

Ein weiteres Beispiel soll demonstrieren, wie man mit dieser Zeigerarithmetik aus einem vordefinierten Datenbereich einzelnen Teile davon für Zeiger und Vektoren zur Verfügung stellen kann. Es zeigt zudem exemplarisch, wie die entsprechenden Funktionen von C, nämlich malloc() und free() arbeiten.

```
#define NULL 0                          /* Fehlerwert             */
#define ALLOCSIZE 1000                  /* gewählte Puffergröße   */
static char allocbuf[ALLOCSIZE];        /* Puffer                 */
static char *allocp = allocbuf;         /* nächster Zeiger        */

char *alloc(n)     /* liefert Zeiger auf Platz für n Zeichen      */
int n;
{
  if (allocp + n <= allocbuf + ALLOCSIZE) {
    allocp += n;                  /* nächsten Zeiger berechnen   */
    return(allocp - n);       /* letzten Zeiger zurückgeben */
  }
  else
    return(NULL);                 /* nicht genügend Platz        */
}

void free(p)       /* Gibt Speicher ab p wieder frei             */
char *p;
{
  if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
    allocp = p;
}
```

Diese beiden Funktionen verwalten den in der Variablen *allocbuf[]* bereitgestellten Speicherplatz in Form eines *last in - first out* - Speichers, wie es in der Regel bei sogenannten Kellerspeichern (*stack*) üblich ist.

Die in C implementierten Funktionen zur Speicherverwaltung benutzen keinen vordeklarierten Speicher wie im Beispiel, sondern greifen auf die entsprechenden DOS-Funktionen zur Zuweisung und Freigabe von Speicher aus dem sogenannten Heap zurück, der im wesentlichen nur durch den insgesamt verfügbaren Rechnerspeicher (nach Abzug des von Betriebssystem, Programmcode und Programmdatensegment benötigten Speichers) begrenzt wird.

6. Strukturen

Eine Struktur bezeichnet einen Datentyp, bei dem eine oder mehrere Variablen, die auch unterschiedliche Datentypen besitzen können, unter einem Typnamen zusammengefaßt und für komplexe Variablen benutzt werden können. In der Sprache Pascal bezeichnet man diese Strukturen als *records*.

6.1 Definition einer Struktur

Grundsätzlich ist eine Struktur eine Typvereinbarung, die mehrere Variablen zusammenfaßt. Sie besteht aus dem Schlüsselwort struct, gefolgt von einem frei wählbaren Bezeichner, der den Typnamen festlegt und einer von geschweiften Klammern umschlossenen Liste von Variablendeklarationen, die unter diesem Typ zusammengefaßt werden sollen:

```
struct typname {
        typ var1;
        typ var2;
        ...
        };
```

In dieser Form wird zunächst lediglich festgelegt, welche Inhalte mit dem mit *typname* bezeichneten Variablentyp verbunden werden sollen, es wird noch keine Variable deklariert und somit auch kein Speicherplatz reserviert.
Als Grundlage für die folgenden Beispiele soll folgende Struktur dienen:

```
struct datum {
  int tag;
  int monat;
  int jahr;
  char m_name[10];
};
```

Hiermit wird ein Typ mit dem Namen *datum* definiert, der die Komponenten *tag*, *monat*, *jahr* und *m_name* (Monatsname) enthält. Während die ersten drei Komponenten vom Typ *int* sind und so jeweils 2 Byte belegen, ist *m_name* ein Array von *char* und belegt somit 10 Byte. Insgesamt benötigt eine Variable, die dem Typ *datum* zugewiesen wird, folglich 16 Byte.
Bei komplexeren Programmen ist es üblich, derartige Strukturdefinitionen in einer .h-Datei zusammenzufassen und in den einzelnen Modulen die entsprechenden Variablen mit einer verkürzten Schreibweise mit diesem Datentyp zu verbinden.

6.2 Deklaration einer Struktur-Variablen

Um eine Variable als Struktur zu deklarieren, ersetzt man den Bezeichner, der den Typ angibt, durch das Schlüsselwort struct und den Namen der definierten Struktur. Um also nach obiger Definition eine Variable d von diesem Typ zu erhalten, schreibt man

```
struct datum d;
```

Die so deklarierte Variable *d* enthält nun mehrere Komponenten, so wie sie zwischen den geschweiften Klammern der Strukturverienbarung aufgelistet wurden und belegt im Speicher einen Platz von 16 Byte Größe. Analog wie bei elementaren Datentypen können durch Kommata getrennt auch mehrere Variablen deklariert werden, z.B.:

```
struct datum d1, d2, d3;
```

Wird eine Struktur nur einmal benötigt, können die Definition und die Deklaration auch zusammengefaßt werden, so würden sich die Variablen des letzten Beispiels auch erzeugen lassen durch die Anweisung

```
struct datum {
  int tag;
  int monat;
  int jahr;
  char m_name[10];
} d1, d2, d3;
```

6.3 Zugriff auf Struktur-Komponenten

Der Zugriff auf die Inhalte einer als Struktur deklarierten Variablen erfolgt durch die Angabe des Variablennamens, gefolgt von einem Punkt und dem in der Strukturvereinbarung definierten Namen der Komponente. Somit bezeichnet

```
d.tag          /* die Komponente tag      */
d.monat        /* die Komponente monat    */
d.jahr         /* die Komponente jahr     */
d.m_monat      /* die Komponente monat    */
```

In diesem Sinne können Komponenten einer Strukturvariablen (bis auf die Schreibweise) behandelt werden wie elementare Datentypen. Das folgende Programm zeigt die Zuweisung und anschließende Ausgabe der Komponenten der Variablen *d* vom Typ *struct datum*:

```
main()
{
struct datum {
     int tag;
     int monat;
     int jahr;
     char m_name[10];
     } d;

     d.tag = 1;
     d.monat = 2;
     d.jahr = 1997;
      strcpy(d.m_name,"Februar");
     printf(" Heute ist der %d.%d.%d im Monat %s\n",d.tag,d.monat,d.jahr,d.m_name);
      return(0);
}
```

In diesem Beispiel wurde die Variable d zusammen mit der Definition der Struktur deklariert, dies hat jedoch den Nachteil, daß es so nicht möglich ist, in anderen Funktionen auf diese Struktur zurückgreifen zu können. Durch eine globale Definition der Struktur, die dann auch in eine .h-Datei verlagert werden kann, können alle assoziierten Module auf diese Definition zurückgreifen und Variablen damit deklarieren:

```
struct datum {
  int tag;
  int monat;
  int jahr;
  char m_name[10];
};

main()
{
  struct datum d;

  d.tag = 1;
  d.monat = 2;
  d.jahr = 1997;
   strcpy(d.m_name,"Februar");
  printf(" Heute ist der %d.%d.%d im Monat %s\n",d.tag,d.monat,d.jahr,d.m_name);
  return(0);
}
```

Wenn z.B. der erste Buchstabe des Monatsnamens, der in *struct datum* als Vektor definiert wurde, ausgegeben werden soll, kann die folgende Anweisung benutzt werden:

```
putch(d.m_name[0]);
```

6.4 Zeiger auf Strukturvariable

Da es sich bei Strukturvariablen um Speicherbereiche handelt, lassen sich auch Variablen verwenden, die einen Zeiger auf einen solchen Speicherbereich repräsentieren. Entsprechend den Vereinbarungen für Zeiger auf elementare Variablen erfolgt auch die Deklarartion eines Zeigers auf eine Struktur durch einen vorangestellten Asterisk '*':

```
struct datum *pd;
```

deklariert einen solchen Zeiger. Der Zugriff auf eine Komponente dieses Zeigers erfolgt analog wie bei Zeigern auf elementare Datentypen durch ein vorangestelltes '*', wobei allerdings durch entsprechende Klammerung eine Rangfolge zwischen den Operatoren '.' und '*' sichergestellt werden muß, da der '.'-Operator stärker bindet als der '*'-Operator.

```
(*pd).tag = 1;
```

bedeutet also, daß *pd* ein Zeiger auf eine Stuktur ist, deren Komponente *tag* der gewünschte Wert zugewiesen werden soll. Um die Schreibweise etwas zu vereinfachen, existiert eine andere Darstellung, die den Zeigercharakter der Struktur stärker hervorhebt:

```
pd->tag = 1;
```

dies bedeutet, daß *pd* ein Zeiger auf die Strukturkomponente *tag* ist, der der Wert 1 zugewiesen werden soll. Die beiden Schreibweisen sind jedoch äquivalent.
Die Operatore '.', '*' und die eckigen Klammern für Vektoren binden stärker als alle anderen Operatoren, besondere Vorsicht ist deshalb geboten bei Inkrement- oder Dekrement-Operatoren. Die Anweisung

```
++pd->tag;
```

erhöht den Inhalt der Komponente *tag*, während die Anweisung

```
(++pd)->tag;
```

den Zeiger erhöht, der anschließend auf eine (neue) Komponente *tag* zeigt, ohne jedoch diese selbst zu verändern.
Entsprechend bewirkt die Anweisung

```
*pd->m_name;
```

einen Zugriff auf das Objekt, auf das *m_name* zeigt und nicht auf das, auf das *pd* zeigt.

6.5 Operationen mit Strukturvariablen

Im vorigen Kapitel wurde der Zugriff auf einzelne Komponenten einer Struktur beschrieben, es sind jedoch auch Operationen mit der gesamten Variablen möglich, wobei die Strukturelemente nicht explizit aufgeführt werden müssen.

6.5.1 Struktur als Funktionsargument

Eine Strukturvariable kann wie jede andere Variable auch als Parameter einer Funktion übergeben werden. Tatsächlich wird dabei auf dem Stack eine Kopie des Inhaltes der Variablen abgelegt, auf die die Funktion zurückgreift. Um die Reihenfolge der Komponenten zu erhalten, muß der Parameter in der Funktion mit dem entsprechenden Typ versehen werden. Änderungen an den Inhalten der so übergebenen Variablen sind aber lediglich lokal während der Laufzeit der Funktion wirksam. Die Ausgabe kann also auch durch Aufruf einer Funktion *output()* erfolgen:

```
...
strcpy(date.m_name,"Februar");
output(date);
...
output(d)
struct datum d;
{
  printf(" Heute ist der %d.%d.%d im Monat %s\n",d.tag,d.monat,d.jahr,d.m_name);
}
```

6.5.2 Struktur als Funktionstyp

Wie bei den elementaren Variablentypen kann eine Funktion auch einen Wert zurückgeben, der nun aber auf eine Struktur verweist. Somit kann eine Strukturvariable dadurch zugewiesen werden, daß sie den Rückgabewert einer entsprechend deklarierten Funktion zugewiesen bekommt, wie das folgende Beispiel zeigt. Man beachte, daß die Rückgabe der Werte wieder über den Stack erfolgt, dessen Größe und Organisation durch die Angabe *struct datum* vor dem Funktionsnamen definiert wird und mit der Struktur der zuzuweisenden Variablen übereinstimmen muß.

```
struct datum input();       /* Verhindert re-definition error   */

main()
{
  struct datum d;
  ...
  d = input();
  ...
}

struct datum input()
{
  struct datum date;

  date.tag = 1;
  date.monat = 2;
  date.jahr = 1997;
  strcpy(date.m_name,"Februar");
  return(date);
}
```

6.5.3 Adreßoperator

Durch den Adreßoperator & kann die Adresse einer Strukturvariablen bestimmt und zum Beispiel an eine Funktion übergeben werden. Diese Funktion ist somit in der Lage, auch die Inhalte der Variablen zu ändern und an die aufrufende Funktion zurückzugeben. Wird an eine Funktion als Parameter die Adresse einer Strukturvariablen übergeben, muß diese nun den '->'-Operator benutzen, um auf die einzelnen Komponenten zuzugreifen:

```
main()
{
  struct datum d;
  ...
  input(&d);
  ...
}

input(date)
struct datum *date;
{
  date->tag = 1;
  date->monat = 2;
  date->jahr = 1997;
  strcpy(date->m_name,"Februar");
  return(0);
}
```

6.5.4 Direkte Zuweisung

Variablen, die mit der gleichen Struktur deklariert wurden, können auch unmittelbar zugewiesen werden, um z.B. den Inhalt einer Strukturvariablen auf eine andere Strukturvariable zu kopieren.

```
struct datum d1, d2;
...
input(&d1);
d2 = d1;
...
```

Nach der Eingabe der Komponenten von *d1* in der Funktion *input()* werden die Inhalte auf die Variable *d2* kopiert. Diese unmittelbare Zuweisung wird z.B. dann benötigt, wenn in einem Sortierverfahren die Inhalte zweier Elemente ausgestauscht werden sollen und eine dritte Komponente als Hilfsvariable benötigt wird:

```
struct datum d1, d2, help;
...
help = d1;
d1 = d2;
d2 = help;
...
```

Soll das Vertauschen der Variablen in einer Funktion erfolgen, muß der Adreßoperator verwendet werden, in der Funktion selbst müssen dann die Inhalte der durch die übergebenen Zeiger definierten Speicherbereiche ausgetauscht werden, wie es die folgende Funktion zeigt:

```
struct datum d1, d2;
...
tausche(d1,d2);
...

tausche(d1,d2)
struct datum *d1, *d2;
{
  struct datum help;
  help = *d1;
  *d1 = *d2;
  *d2 = help;
}
```

Auch hier ist es nicht nötig, die Komponenten einzeln zu behandeln. Anders als beim komponentenweisen Zugriff müssen Klammern hier nicht gesetzt werden, da nur ein Operator eingesetzt wird.

6.5.5 Vektoren von Strukturvariablen

Strukturen werden besonders häufig verwendet, um in Form von Vektoren logisch zusammenhängende Variablen zu verwalten. Um z.B. Monatsnamen und ihre Kurzformen zu sichern, ist folgende Vektorstruktur geeignet, wobei auch gleich gezeigt wird, wie eine Deklaration als Konstante vorgenommen werden kann:

```
struct monatsname {     /* Die Struktur heißt monatsname, sie     */
  char *kurzname;         /* enthält zwei Zeiger-Komponenten mit  */
  char *langname;         /* den nachfolgend zugewiesenen Strings.*/
} M_Name[] = {            /* Die Variable heißt M_Name, die       */
  {"Jan", "Januar",   }   /* Länge des Vektors wird automatisch   */
  {"Feb", "Februar", }    /* aus der Anzahl der Einträge errech-  */
  {"Mrz", "März",     }   /* net. Die einzelnen Komponenten wer-  */
  ...                     /* den durch Kommata getrennt aufgeli-  */
  {"Dez", "Dezember" }    /* stet, jeder Vektoreintrag wird mit   */
}                         /* geschweiften Klammern verbunden.     */
```

7. Dateien

Datenverarbeitung im weitesten Sinne setzt voraus, daß die zu bearbeitenden Daten auch außerhalb der Laufzeit eines Programmes - z.B. nach Abschalten des Rechners - wieder zur Verfügung gestellt werden können, wenn das Programm erneut gestartet wird. In jedem Fall bedient man sich dazu einer seuqmetiellen methode, in der die Daten bitweise auf ein geeignetes Medium gesichert werden. In früheren Zeiten benutzte man dazu Löcher in Papierstreifen oder Lochkarten, später nutzte man tonbandähnliche Bänder, diese wurden abgelöst durch magentisierbare Scheiben, wie sie in Festplatten-laufwerken und Disketten zu finden sind, und heute ist man wieder auf die schon früher benutzten Löcher zurückgegekommen, die man in dünne Folien brennt und mit einem Laserstrahl wieder abtastet (CD-ROM).
Aus der Sicht eines Programmes stellt sich jegliche Datensicherung immer als ein Byte-Strom dar, weshalb gerade in der Sprache C häufig von einem *streamed I/O* (Input/Output) gesprochen wird.

7.1 Datentyp FILE

Um Daten in einer Datei ablegen zu können, ist eine besondere Strukturvariable nötig, die den Dateinamen des Benutzers mit einer physikalischen Datei auf einem Datenträger verbindet. Das Betriebssystem stellt dazu einen Speicherbereich zur Verfügung, der einerseits Informationen über die Datei enthält, wie z.B. Datum und Uhrzeit der letzten Änderung, Größe der Datei, Zugriffsberechtigungen usw., zum anderen einen ausreichend großen Speicherbereich - den sogenannten Puffer - bereitstellt, über den die Informationen zwischen Datei und Programm ausgetauscht werden können. Diese Strukturvariable wird als ein Zeiger deklariert, ihre Definition erfolgt in der Standardbibliothek*stdio.h*.
Um in einem Programm mit Dateien arbeiten zu können, muß eine Variable deklariert werden, die diesen Datenaustausch ermöglicht, dabei übernimmt *FILE* die Bedeutung eines eigenständigen Datentyps:

```
FILE *fp;
```

Man bezeichnet wgen seiner Verwandschaft mit Zeigern die so zugewiesene Variable häufig auch als einen Filepointer.

7.1.1 *FILE* ***fopen(****const char *name, const char *modus)*

Die Verbindung eines Filepointers mit einer Datei erfolgt mit einer von C bereitgestellten Funktion *fopen()*, die einen Zeiger auf eine passende Struktur zurückgibt, wenn sie erfolgreich eine Verbindung zu einer Datei hat aufbauen können, ansonsten gibt sie den ebenfalls in*stdio.h* vordefinierten Wert*NULL* zurück. Die Funktion benötigt zwei Parameter: Der erste Parameter gibt den Namen der Datei an, zu der eine Verbindung hergestellt werden soll, der zweite Parameter bestimmt die Zugriffsart, mit der die Datei bearbeitet werden soll. Name und Zugriffsart werden durch Zeichenketten angegeben. Für die Zugriffsart gibt es nur die drei folgenden Alternativen:

"r"	(read)	Die Datei wird nur zum, Lesen geöffnet
"w"	(write)	Die Datei wird nur zum Schreiben oder zur Neuanlage geöffnet
"a"	(append)	An eine Datei werden weitere Daten angefügt bzw. die Datei neu angelegt

Der Parameter für den Dateinamen muß einen gültigen Suchweg und Dateinamen enthalten, damit die Verbindung hergestellt werden kann. Im folgenden Beispiel für die Zuweisung eines Filepointers wird eine Datei *test.txt* im Verzeichnis *C:\texte* zum Lesen geöffnet und anschließend geprüft, ob die Verbindung hergestellt werden konnte:

```
FILE *txt;
...
txt = fopen("C:\texte\test.txt","r");
if (txt == NULL) fehlermldg();
...
```

Durch Zusammenfassen der Vorgänge Öffnen und Überprüfung ergibt sich die verkürzte Darstellung

```
...
if ((txt = fopen("C:\texte\test.txt","r")) == NULL) fehlermldg();
...
```

Man beachte, daß*txt* unbedingt mit dem Wert*NULL* (und nicht mit dem Zahlenwert 0) verglichen wird, wenn die Kompatibilität zu anderen Compilern und Betriebssystemen gewährleistet sein soll.
Soll die Datei zum Schreiben geöffnet werden, muß nur die Zugriffsart geändert werden:

```
...
if ((txt = fopen("C:\texte\test.txt","w")) == NULL) fehlermldg();
...
```

Achtung: Existiert bei diesem Aufruf bereits eine Datei mit dem angegebenen Namen, so wird sie durch diesen Aufruf gelöscht und neu angelegt - ihr voriger Inhalt ist unwiederbringlich verloren!

Sollen an eine bereits bestehende Datei Daten angefügt werden, ohne die Datei zu löschen, ist die Zugriffsart *append* zu benutzen. Sie öffnet die Datei wie zum Lesen, stellt den Datenzeiger auf eine Position hinter dem letzten Eintrag in der Datei und schaltet um in die Zugriffsart*write*, wie im folgenden Beispiel gezeigt:

```
...
if ((txt = fopen("C:\texte\test.txt","a")) == NULL) fehlermldg();
...
```

Mit keiner dier hier gezeigten Methoden ist es möglich, bereits geschriebene Daten nachträglich in einer Datei zu ändern, dazu müssen Hilfskonstruktionen herangezogen werden, die sich die Möglichkeit zu eigen machen, daß ein Programm mehrere Dateien gleichzeitig öffnen und mit jeweils einer eigenen Zugriffsart bearbeiten kann.

*7.1.2 void **fclose(**FILE *stream**)***

Bei Schreibvorgängen wird in der Regel nicht jedes Zeichen einzeln in die Datei geschrieben, sondern das Betriebssystem sammelt eine bestimmte Anzahl von zu schreibenden Zeichen in einem Puffer und schreibt dessen Inhalt in die Datei, wenn der Puffer gefüllt ist oder eine explizite Anweisung zum Leeren des Puffers erfolgt. Bei einem normalen Programmende wird der Puffer beim Beenden des Programmes geleert, aber ein vorzeitiger Abbruch - z.B. durch eine Fehlerbedingung - oder das erneute Öffnen der gleichen Datei kann dazu führen, daß die im Puffer befindlichen Daten nicht mehr in die Datei übertragen wurden. Um das Leeren des Puffers zu bewirken und die Vwerbindung zu der Datei aufzulösen, dient die Funktion *fclose()*:

```
...
fclose(txt);
...
```

Nach diesem Aufruf ist der Puffer vollständig an die Datei übertragen und geleert und die Verbindung zur Datei unterbrochen.

*7.1.3 void **fflush(**FILE *stream**)***

Soll - z.B. aus Sicherheitsgründen - lediglich der Inhalt des Puffers in die Datei übetragen, aber die Verbindung noch nicht unterbrochen werden, verwendet man den Befehl *fflush()*, er stellt sicher, daß auch bei einem Fehlerabbruch des Programmes oder einem Stromausfall die Daten in der Datei gespeichert worden sind:

```
...
fflush(txt);
...
```

*7.1.4 void **fcloseall()***

Wurden von einem Programm mehrere Dateien geöffnet, können mit dieser Funktion alle auf einmal geschlossen und ihr restlicher Inhalt vom Puffer auf die jeweilige Datei übertragen werden:

```
...
fcloseall();
...
```

7.2 Zeichenorientierte Ein-/Ausgabe bei Dateien

Wie Pascal kennt C einen Datentyp, der auf dem Datentyp *char* basiert, der jeweils ein Zeichen auf dem Datenträger repräsentiert. Mit ihm lassen sich am einfachsten Texte abspeichern, da jedem Byte eindeutig ein Zeichen zugeordnet ist. Voraussetzung für das Arbeiten mit einer solchen Datei ist immer ein erfolgreich zugewiesener Filepointer, wobei nur die Aktionen zulässig sind, die beim Öffnen der Datei vereinbart wurden.

*7.2.1 char **getc(**FILE *stream**) - Einlesen eines Zeichens von einer Datei***

Wie bereits in 3.2.4 (S. 16) beschrieben, liefert diese Funktion aus einem Dateistrom ein einzelnes Zeichen zurück. Ist das Dateiende erreicht, wird das Zeichen *EOF* übergeben, das somit eine eigene Bedeutung besitzt und als einziges Zeichen nicht mitten in der Textdatei auftreten darf. Andere Steuerzeichen wie *CR* oder *LF* werden wie normale Zeichen behandelt und übergeben. Das folgende Beispiel öffnet eine Datei *test.txt* und gibt den Inhalt zeichenweise auf dem Bildschirm aus, bis das Dateiende erreicht ist:

```
#include <stdio.h>
main()
{
  FILE *txt;
  char c;

  if ((txt = fopen("test.txt","r")) == NULL) return(-1);
  while ((c = getc(txt)) != EOF) putch(c);
  return(0);
}
```

Häufig wird statt der Zeichenfolge CR-LF lediglich ein LF-Zeichen (Zeilenvorschub) in der Datei abgelegt. Damit die Ausgabe der nächsten Zeile dennoch am Anfang der Zeile beginnt, muß das obige Programm statt der kursiv gedruckten Zeile folgende Zeilen erhalten:

```
while ((c = getc(txt)) != EOF) {
  if (c == 0x0a) putch(0x0d);       /* 0x0a = LF, 0x0d = CR   */
  putch(c);
}
```

wobei vor jeder Ausgabe eines LF-Zeichens zusätzlich ein Wagenrücklauf CR ausgegeben wird.

*7.2.2 char **putc(**char c, FILE *stream**) - Ausgabe eines Zeichens in eine Datei***

Die Funktion putc() wurde bereits in 3.1.3 (S. 13) kurz beschrieben. Sie legt ein Zeichen in einer Datei ab, die zuvor erfolgreich geöffnet und im Modus *write* oder *append* betrieben wird. Die meisten Texteditoren beenden eine Zeile lediglich mit einem Linefeed-Zeichen (LF), während von der Tastatur bei Betätigen der Eingabetaste ein Carriage-Return-Zeichen (CR) erzeugt wird. Damit eine direkt von der Tastatur eingegebene Datei richtig ausgegeben wird, muß also jedes CR-Zeichen in ein LF-Zeichen umgewandelt werden, wie es im folgenden Beispiel geschieht (hier ist nur der Anweisungsteil wiedergegeben, der Funktionskopf ist der selbe wie im oberen Beispiel):

```
...
    if ((txt = fopen("test_wr.txt","w")) == NULL) return(-1);
    while ((c = getch()) != 0x1a) {
        if (c != 0x0d) putc(c,txt); else putc(0x0a,txt);
        putch(c);
    }
    fclose(txt);
    return(0);
```

*7.2.3 char **fgets(**char *string, int n, FILE *stream**) - Einlesen einer Zeichenkette von Datei*

Eine Zeichenkette wird in C mit einem Null-Character abgeschlossen, während in einer Textdatei üblicherweise ein Linefeed-Zeichen als Ende einer Zeile eingesetzt wird. Die Funktion*fgets()* liest in der Datei ab der aktuellen Position solange Zeichen in die Zeichenkette *string* ein, bis einer der folgenden drei Fälle eintritt:

- das Linefeed-Zeichen wird gelesen
- das Dateiende wird erreicht
- es wurden bereits n Zeichen gelesen

In allen Fällen wird die Zeichenkette durch ein Null-Character ergänzt. Jedes andere bis dahin gelesene Zeichen, also auch das LF-Zeichen am Zeilenende, sind im*string*enthalten. Das folgende Beispiel nutzt diese Funktion, um den Inhalt einer Textdatei zeilenweise auszugeben (Da die hier verwendete Ausgabefunktion*puts()* selbst jeweils einen Zeilenvorschub erzeugt, erscheint zwischen jeder Ausgabezeile eine zusätzliche Leerzeile):

```
#include <stdio.h>
main()
{
     FILE *txt;
     char c;
     char *string;

     string = (char *) malloc(255);

     if ((txt = fopen("test.txt","r")) == NULL) return(-1);
     while ((string = fgets(string,255,txt)) != NULL) puts(string);
     return(0);
}
```

*7.2.4 int **fputs(**char *string, FILE *stream**) - Ausgeben einer Zeichenkette in eine Datei*

Die Funktion *fputs()* gibt eine Zeichenkette in die Datei aus. Dabei wird zwar das abschließende 0-Character entfernt, dafür aber kein LF-Zeichen angefügt, sodaß durch eine zusätzliche Zeichenausgabe, z.B. mit *fputc()* das entsprechende Zeichen in die Datei geschrieben werden muß, wie es im folgenden Beispiel gezeigt wird. Die Eingabe von Zeilen wird durch eine Leerzeile beendet:

```
#include <stdio.h>
main()
{
  FILE *txt;
  char c;
  char *string;

  string = (char *) malloc(255);

  if ((txt = fopen("test_wr.txt","w")) == NULL) return(-1);
  do {
    gets(string);
    fputs(string,txt);        /* Zeichenkette speichern */
    fputc(0x0a,txt);          /* LF-Zeichen speichern   */
  }
  while (string[0] != '\0');  /* Ende mit leerer Zeile  */
  fclose(txt);
  return(0);
}
```

Inhaltsverzeichnis