Grundlagen der Informatik

Vorlesungsskript

Prof. Dr. T. Gervens, Prof. Dr.-Ing. B. Lang, Prof. Dr.-Ing. C. Westerkamp

4 EINFÜHRUNG IN C 2

4.1 WISSENSWERTES ÜBER C 2
4.2 ERSTE BEISPIELE 2
4.3 DARSTELLUNG VON PROGRAMMCODE DURCH STRUKTOGRAMME 4
4.4 VARIABLEN 5
4.5 KONSTANTEN 7
4.6 EIN- UND AUSGABE VON ZEICHEN 8
4.7 ARITHMETISCHE OPERATIONEN 8
4.8 MATHEMATISCHE FUNKTIONEN 9
4.9 WERTZUWEISUNG 9
4.10 AUSGABE-ANWEISUNG MIT PRINTF 10
4.11 EINGABEANWEISUNG MIT SCANF 12
4.12 VERGLEICHSOPERATOREN 13
4.13 KONTROLLSTRUKTUREN 14
4.13.1 SEQUENZ, VERBUNDANWEISUNG 14
4.13.2 EINFACHE IF-ELSE ABFRAGE 14
4.13.3 MEHRFACHABFRAGE - ELSE IF 17
4.13.4 MEHRFACHABFRAGE - SWITCH 18
4.13.5 WHILE - SCHLEIFE 19
4.13.6 DO-WHILE - SCHLEIFE 20
4.13.7 FOR - SCHLEIFE 20
4.14 STEUERUNG VON SCHLEIFEN DURCH BREAK UND CONTINUE 22
4.15 LOGISCHE OPERATOREN 23
4.16 INKREMENT- UND DEKREMENT 24
4.17 ZUWEISUNGSOPERATOREN 25
4.18 SHIFTOPERATOREN 25
4.19 BIT-OPERATOREN 25
4.20 FRAGEZEICHENOPERATOR 26
4.21 BINDUNGSSTÄRKEN VON OPERATOREN 27
4.22 FELDER 28
4.23 ZEICHENKETTEN 29
4.24 TIPPS UND TRICKS ZUM LÖSEN DER AUFGABEN 33

4 Einführung in C

4.1 Wissenswertes über C

- C ist eng mit UNIX verbunden, 90% von UNIX ist in C geschrieben
- Entstanden aus den ersten UNIX-Sprachen (ca. 1970) aus der Sprache B
  Entwickler: Ken Thompson
  Dennis Ritchie
- C ist eine vollständig typisierte Programmiersprache. Dies bedeutet, dass jedes Datenobjekt einem eindeutigen Typ zugeordnet werden muss (z.B. Zeichen, Ganze Zahl, Gleitkommazahl, ...). Dieser Typ bestimmt, welche Werte diese Variable annehmen darf.
- Standard von C war lange Zeit das von Kernighan/Ritchie herausgegebene Buch "The C Programming Language".
- 1988 wurde ANSI C veröffentlicht

4.2 Erste Beispiele

```
main() {
}
```

Dieses Programm besteht aus einer Funktion namens **main**, was an den runden Klammern deutlich wird. In den geschweiften Klammen steht der Funktionsrumpf {....}, der in diesem Fall leer ist.

Jedes C-Programm muss eine main-Funktion enthalten. Von dieser main-Funktion aus werden alle weiteren Programmteile aufgerufen, welche auch in Funktionsform dargestellt werden.

Im Grunde ist jedes C-Programm eine Aneinanderreihung und Verschachtelung von Funktionen, also ein Funktionenbaum.

Obiges Programm kann mit einem Texteditor als ASCII-Datei **`test.c`** erstellt werden. Von der Betriebssystemumgebung aus kann man den C-Compiler aufrufen, um das Quellprogramm in Maschinencode zu übersetzen.

```
cc test.c ↵
```

Der Compiler schreibt den Maschinencode in eine Datei **`a.out`**.

Das Programm wird ausgeführt durch

```
a.out ↵
```

Mit

```
cc test.c -o test.exe
```

schreibt der Compiler den Maschinencode in eine Datei namens **`test.exe`**.

Enthält das Programm Fehler, schreibt der Compiler diese Fehler auf den Bildschirm (Standard-Fehlerausgabe).

Obiges Programm tut überhaupt nichts. Damit im Programm etwas passiert sind im Funktionsrumpf **Anweisungen** einzugeben.

Die einfachste Anweisung ist die leere Anweisung, die auch nichts bewirkt:

```
main() {
   ; /* leere Anweisung */
}
```

Zwei Dinge sind jetzt zu beachten:

- Hinter jeder Anweisung steht ein **Semikolon**!!!
- **Kommentare** werden zwischen /* und */ eingeschlossen. Die Kommentarzeichen und alle Zeichen zwischen den Kommentarzeichen werden vom Compiler ignoriert. Kommentare sind nicht auf eine Zeile beschränkt.

```
/*
Dies
ist auch
ein
Kommentar
*/
```

Ein Beispiel das tatsächlich etwas bewirkt:

```
#include <stdio.h>
main(){
   printf ("hello, world");
}
```

Dieses Programm bringt den Text:

```
hello, world
```

auf den Bildschirm. Dies geschieht folgendermaßen: Die Hauptfunktion ***main*** ruft die Bibliotheksfunktion ***printf*** mit dem Argument "hello, world" auf. Durch ***#include <stdio.h>*** wird die Datei ***stdio.h*** vor dem Compilieren hinzugefügt. In dieser Datei stehen notwendige Informationen zu ***printf***.

Erweiterung des Programms:

```
#include <stdio.h>
main(){
   printf ("hello, world \n");
}
```

Die Zeichenkette ***\n*** ist die C-Schreibweise für einen Zeilentrenner. Die Ausgabe wird dann am linken Rand der neuen Zeile fortgesetzt.

Beachte: ***\n*** repräsentiert nur ein Zeichen

Analog: ***\t*** (Tabulatorsprung)

Rufen wir die Funktion printf mit folgendem Agument auf:

```
printf ("\nh\ne\nl\nl\no\n");
```

führt zu folgender Ausgabe:

```
h
e
l
l
o
_ (←Cursor)
```

4.3 Darstellung von Programmcode durch Struktogramme

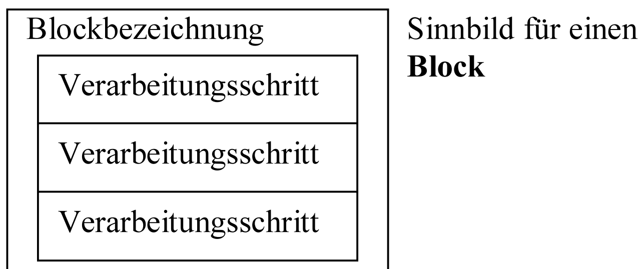
Zum Entwurf eines Programms ist es hilfreich, nicht sofort mit dem Programmieren zu beginnen, sondern den Ablauf des Programms zunächst zu visualisieren. Für diesen Schritt sind **Struktogramme** sehr gut geeignet, die nach ihren Urhebern oftmals auch als **Nassi-Schneidermann-Diagramme** bezeichnet werden.

Struktogramme führen dazu, dass man ein Problem strukturiert in Teilprobleme zerlegt. Sprunganweisungen (GOTO) sind dabei nicht erlaubt. Als Mittel erlaubt die strukturierte Programmierung vielmehr die **Sequenz**, die **Iteration** und die **Selektion**. Die Elemente der Struktogramme werden im folgenden zusammen mit den zugehörigen C-Konstrukten eingeführt.

Das Grundelement des Struktogramms ist ein **Verarbeitungsschritt**, welcher als ein Rechtecksymbolisch dargestellt wird. Im Inneren des Rechtecks steht eine Beschreibung des Verarbeitungsschritts.

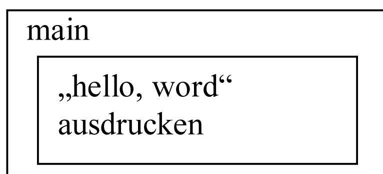
Es ist es dem Programmierer überlassen, wie komplex er seine Verarbeitungsschritte wählt. In einer Übersicht kann beispielsweise in einem einzelnen Verarbeitungsschritt der Großteil eines Programms zusammengefasst werden, in einer Feinspezifikation wird hingegen jede einzelne C-Anweisung ein Verarbeitungsschritt dokumentiert.

Zur Kombination mehrerer zusammenhängender Verarbeitungsschritte dient der **Block**.



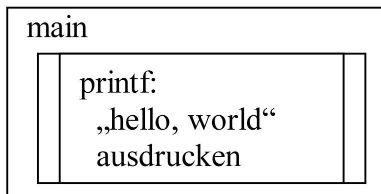
Er kann einem Haupt- oder Unterprogramm entsprechen, aber auch einfach mehrere Verarbeitungsschritte unter einer gemeinsamen Bezeichnung zusammenfassen.

Damit kann das bisher behandelte Programm wie folgt dargestellt werden:



In dieser Darstellung wird nicht berücksichtigt, dass der Ausdruck durch das Unterprogramm „printf“ durchgeführt wird. Möchte man dies aufzeigen, benötigt man ein Sinnbild für einen Unterprogrammaufruf:

Damit lässt sich die Beschreibung des Programms noch detaillieren:



Es ist dem Programmierer beim Entwurf des Struktogramms überlassen, welchen Detaillierungsgrad er für die Beschreibung wählt. Der Detaillierungsgrad sollte so gewählt werden, dass die im Programm realisierten Algorithmen und ihre Implementierung verständlich werden und damit der zugeordnete C-Code dokumentiert wird.

4.4 Variablen

Variablen im Programm werden über die Vergabe eines Namens vereinbart. Für die Gültigkeit eines **Variablennamen** gilt:

- Besteht aus eine Folge von Ziffern, Buchstaben und dem Unterstrich „_“
- Maximale Länge: 31 Zeichen
- Das erste Zeichen muss ein Buchstabe oder der Unterstrich sein, keine Ziffer
- Es wird zwischen Groß- und Kleinschreibung unterschieden (case sensitive)
- Reservierte Wörter wie **`if`**, **`else`**, **`for`**, **`while`** dürfen nicht verwendet werden.
- Der Name muss eindeutig sein.

Zulässige Namen sind z. B.: **`a`**, **`B`**, **`SUM`**, **`x2`**, **`x_a`**, ...

Nicht zulässig: **`1a`**, **`a-wert`**

Jeder Variablen ist ein **Datentyp** zugeordnet (**`int, float, double, char`**), welcher festlegt, wie der Wert der Variablen im Speicher als Bitmuster abgelegt wird.

`int:` Ganzzahliger Wert (mit VZ)
Darstellung im 2er-Komplement
Länge: üblicherweise 32 Bit (PC häufig 16 Bit)
Die Länge ergibt sich aus der Datenwortbreite des Prozessors.

`float:` Endliche Dezimalzahlen im Gleitkommaformat (mit VZ)
Länge: 32 Bit

`double:` endliche Dezimalzahl im Gleitkommaformat (mit VZ)
Länge 64 Bit (ANSI-C macht keine Vorgaben zur Länge. **`double`** muss nur größer oder gleich **`float`** sein).

`char:` Darstellung eines Zeichens, 8 Bit
Mit dem char-Wert kann auch gerechnet werden. Der ASCII-Code des Zeichens wird dann als Dualzahl interpretiert.

Alle Variablen in C müssen, bevor sie benutzt werden, deklariert/definiert werden. Dabei wird der Typ festgelegt. Variablendeklarationen haben folgende Form:

```
Datentyp v1, v2, v3;
```

Beispiele:

```
int i, j;
float Summe, Nenner;
double Celsius, Grad;
char Zeichen, S;
```

Neben den Grundtypen ***char***, ***int***, **float** und **double** gibt es die Qualifier

short und **long**

bzw.

signed und **unsigned**,

so dass sich folgende Typen definieren lassen:

Typ	Länge in Bytes	Wertebereich	Bedeutung	Bemerkungen
short int oder **short** **unsigned short int**	2 2	$-2^{15} \dots 2^{15}-1$ $0 \dots 2^{16}-1$	Ganzzahl mit Vorzeichen Ganzzahl ohne Vorzeichen	1, 3
int **unsigned int**	4	$-2^{31} \dots 2^{31}-1$ $0 \dots 2^{32}-1$	wie **short** (ANSI)	2, 3
long int oder **long** **unsigned long int**	4 4	$-2^{31} \dots 2^{31}-1$ $0 \dots 2^{31}-1$	Ganzzahl mit Vorzeichen Ganzzahl ohne Vorzeichen	1,3
float	4	$\pm 3.4 * 10^{-38} \dots \pm 3.4 * 10^{38}$	Gleitkommazahl (24+8 Bits), 7 Stellen	3
double **long double**	8 16	$\pm 1.7 * 10^{-308} \dots \pm 1.7 * 10^{308}$ $\pm 3.4 * 10^{-4932} \dots \pm 1.1 * 10^{4932}$	Gleitkommazahl (53+11 Bits), 15 Stellen Gleitkommazahl (65+15Bits) 30 Stellen	3
char **unsigned char**	1 1	-128...127 0 ... 255	ASCII-Zeichen bzw. 8 Bit-Ganzzahl mit Vorzeichen ohne Vorzeichen	3

Bemerkungen:

1. **long int** und **short int** können mit **long** bzw. **short** abgekürzt werden.
2. Die tatsächliche Anzahl von Bits variiert beim **int**-Datentyp je nach Rechnersystem.
3. Mit der Funktion **sizeof** kann die tatsächliche Speicherplatzgröße eines Datentyps in Byte ermittelt werden, z.B.:

```
int a,b;
a=sizeof(unsigned int)
b=sizeof(a);
```

Beispiele:

```
int i=5, gamma=3;
unsigned short int use;
long k;
float a, s, d, f;
char a, zeichen;
```

4.5 Konstanten

Ganzzahlige Konstanten

- Gewohnte Schreibweise `0`, `-23`, `25786`.
- Sie werden im Speicher im **`int`-Format** abgelegt.
- **`long`**-Konstanten werden am Ende mit dem Buchstaben l oder L geschrieben.
- **`Unsigned`** (vorzeichenlose) Konstanten werden am Ende mit u oder U geschrieben.
- Ganzzahlige Konstanten können auch **oktal (0...)** oder **hexadezimal (0x...)** oder **(0X...)** geschrieben werden.

Konstanten für Gleitpunktzahlen

- Dezimal oder Exponentialform: `12.34`, `-1E3`, `.25`, `7E+25`.
  Achtung: Im deutschen Sprachgebrauch wird eins Komma zwei verwendet, in C trennt jedoch der Punkt die Vor- und Nachkommastellen.
- Die Darstellung im Speicher erfolgt im **`double`-Format**.
- Typ **`float`** wird mit der Endung f oder F bezeichnet.

Char-Konstanten

- In einfachen Hochkommata eingeschlossene Zeichen wie `'x'`, `'O'`, `'5'`, `'\n'`, `'\0'`.
  Man beachte `'1'` ist nicht `1`!
- Die Darstellung im Rechner erfolgt als ASCII-Zeichen.

Sonderzeichen für char-Konstanten

'\n'	Zeilenvorschub
'\f'	Seitenvorschub
'\t'	Tabulatorsprung
'\b'	Rücktaste
'\\'	Schrägstrich
'\''	einfaches Hochkomma
'\"'	doppeltes Hochkomma
'\ooo'	Zeichencode in oktaler Darstellung
'\xhh'	Zeichencode in hexadezimaler Darstellung

Beispiele:

Konstante	Art	intern
195	dezimale Ganzzahl	**int**
195l	dezimale Ganzzahl	**long int**
195u	dezimale Ganzzahl	**unsigned int**
0303	oktale Ganzzahl	**int**
0XA3	hexadezimale Ganzzahl	**int**
0XFUL	hexadezimale Ganzzahl	**unsigned long int**
1.25	Gleitkomma	**double**
1.25f	Gleitkomma	**float**
'D'	1 Zeichen	**char**
'\x41'	1 Zeichen	**char**
'\101'	1 Zeichen	**char**

4.6 Ein- und Ausgabe von Zeichen

Die Standardbibliothek **stdio.h** enthält zwei weitere wichtige Funktionen zur Ein- und Ausgabe über Tastatur:

getchar und **putchar.**

getchar liefert bei jedem Aufruf das nächste Zeichen vom Eingabe-Zeichenstrom als Funktionswert. Nach der Anweisung:

```
int c;
c=getchar( );
```

enthält die Variable c das nächste Zeichen. Die Eingabezeichen werden in der Regel von der Tastatur gefordert. Auch Sonderzeichen wie Zeilenumbruch, Tabulator, Ende der Eingabe werden wie einzelne Zeichen behandelt.
Alle gültigen Zeichen, die von **getchar** geliefert werden, lassen sich in einem Byte kodieren und könnten somit in einer Variablen vom Typ **char** gespeichert werden. Das Zeichen "Ende der Eingabe" (End Of Transmission/**EOT**), welches mit **Control-d** auf der Tastatur ausgelöst wird, bewirkt, dass **getchar** danach nur noch den Wert **EOF** liefert. **EOF** ist eine symbolische Konstante, welche das Ende des Eingabestroms signalisiert. Sie ist nicht als Zeichencode vereinbart. Sie wird daher zusätzlich zu den gültigen Zeichencodes als Wert im **int**-Bereich vereinbart. Üblicherweise hat die Konstante **EOF** den Wert -1.
Mit der Anweisung:

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

wird das Zeichen, dessen Code in der der Variablen c gespeichert ist, am Bildschirm ausgegeben.
Beispiel: Kopierprogramm

```
#include <stdio.h>
main(){
   int c;
   c=getchar ();
   while (c!=EOF) {
      putchar (c);
      c=getchar ();
   }
}
```

4.7 Arithmetische Operationen

a+b	Addition
a-b	Subtraktion
a*b	Multiplikation
a/b	Division
a%b	Rest bei ganzzahliger Division (Datentypen: **int**) z. B.: **20%3** = 2
±a	Vorzeichen

Rangordnung der Operationen

+, -	Vorzeichen
***, /, %**	Punktrechnung
+, -	Strichrechnung

Vor Ausführung einer arithmetischen Operation finden Typanpassungen (englisch: Casting) statt, wenn die Operanden von unterschiedlichem Typ sind. Es wird immer vom "niedrigen" in den "höheren" Typ umgewandelt:

char → int → float → double

Vorsicht: Das Ergebnis hat immer den gleichen Typ wie die (angepassten) Operanden. Beim nachfolgenden Beispiel

```
int n;
double x;
n=5;
x= 3/n;
```

erhält somit x den Wert 0!

4.8 Mathematische Funktionen

sqrt(x)	$\sqrt{x}$
exp(x)	e^x
log(x)	ln(x)
log10(x)	$\log_{10}(x)$
pow(x,y)	x^y
sin(x)	sin(x)
cos(x)	cos(x)
tan(x)	tan(x)
asin(x)	arcsin(x)
acos(x)	arccos(x)
atan(x)	arctan(x)
fabs(x)	|x|
sinh(x)	sinh(x)
cosh(x)	cosh(x)

Damit der Compiler diese Funktion findet, ist die entsprechende Informationsdatei **math.h** im Vorfeld zu laden durch:

```
#include <math.h>
```

Als Argumente können Variablen oder auch arithmetische Ausdrücke an die Funktionen übergeben werden. Das Ergebnis der gelisteten Funktionen ist in jedem Fall vom Typ **double**!! Dies muss unbedingt bei der Weiterverarbeitung oder der Ausgabe berücksichtigt werden, sonst entstehen falsche Ergebnisse.

Beispiel:

```
#include <math.h>
...
double x=1.0, z;
z=cos(x*x+tan(x));
```

Wichtig: Ein Programm mit mathematischen Funktionen muss mit

```
cc datei.c -lm
```

übersetzt und gelinkt werden. Hierdurch wird dem Compiler mitgeteilt, in welcher Bibliothek der Objektcode der mathematischen Funktion zu finden ist.

4.9 Wertzuweisung

Die Wertzuweisung in C erfolgt mit dem Operator "=". Die Wertzuweisung hat die folgende Form:

```
Variable=Ausdruck;
```

Der Datentyp des Ausdrucks wird nach der Berechnung umgewandelt in den Datentyp der Variablen!
Erfolgt beispielsweise eine Zuweisung eines Gleitkomma-Ausdrucks auf eine Ganzzahl-Variable, dann werden bei der Umwandlung von **`float`** oder **`double`** nach **`int`** alle Nachkommastellen weggelassen und nur der ganzzahlige Anteil wird der Variablen zugewiesen.
Die Zuweisung endet mit einem Semikolon.

Beispiel: Berechnung der Differenz zwischen sin(x) und $x - \frac{x^3}{3!}$

```
#include <stdio.h>
#include <math.h>
main(){
   float x;
   x=3;
   printf ("Differenz: %f \n",
           sin(x) - x + x * x *x/6);
}
```

oder:

```
#include <stdio.h>
#include <math.h>
main(){
   float x;
   double z;
   x=3;
   z= sin(x) - x + x * x *x/6;
   printf ("Differenz: %f \n",z);
}
```

Innerhalb der Funktion **`printf`** wird in obigem Beispiel ein **`float`** bzw. **`double`**-Wert ausgegeben. Die Position wird durch die Formatangabe „**`%f`**“ festgelegt.

4.10 Ausgabe-Anweisung mit printf

Die Anweisung

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

schreibt den zwischen den beiden Hochkommas "...." stehenden Ausgabetext **`Text`** auf den Bildschirm.

Sonderzeichen wie Zeilenumbruch **`\n`** oder Tabulator **`\t`** können in den Text eingestreut werden. Ein beliebiges Zeichen (und damit auch ein nichtdruckbares Zeichen) kann durch seinen ASCII-Code angegeben werden, z.B. in hexadezimaler Form **`\xhh`** (zweistellige Hexadezimalzahl). Mit

```
printf ("\x07");
```

erzeugt der Rechner beispielsweise einen "Piep", (das ASCII-Zeichen BEL).

In den Ausgabetext kann der Wert von **Variablen** und/oder **Ausdrücken** eingesetzt werden. Die Position wird durch die Stellung von **Formatangaben** festgelegt.

Beispiel:

```
float x=5;
int k=4, anz;
...
anz=printf ("x=%f,  k=%d\n", sin(x), k);
printf ("%d Zeichen gedruckt.\n",anz);
```

Die allgemeine Syntax der **`printf`**-Funktion lautet:

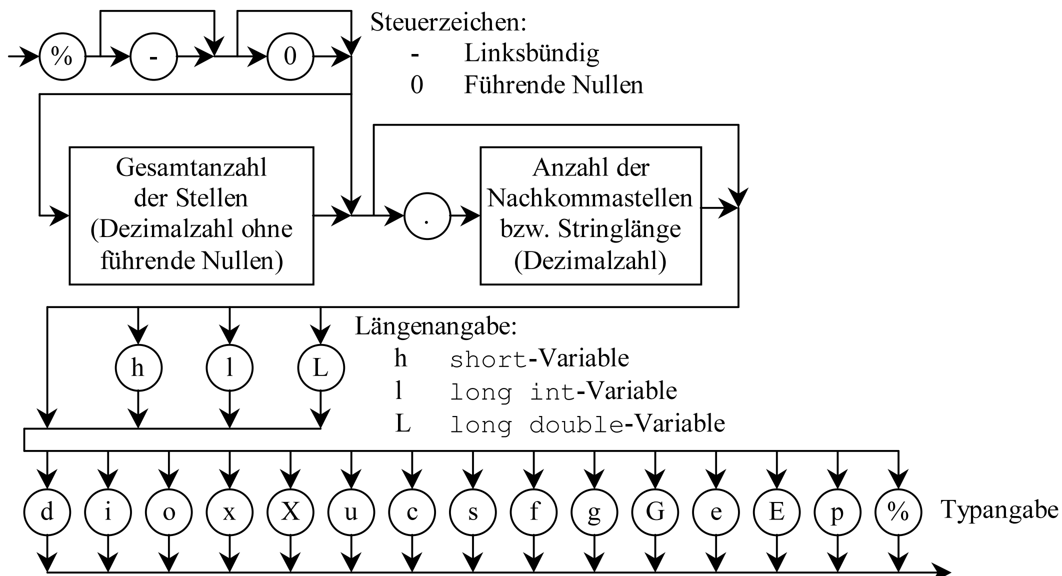
*[**`int`**-Variable=]* **`printf`***(Zeichenkette[,Variable1,Variable2,...,VariableN]);*

Optionale Elemente sind in eckigen Klammern eingeschlossen.

Argumente: Die als Argument übergebene Zeichenkette wird durch die **`printf`**-Funktion auf dem Bildschirm ausgegeben. In die Zeichenkette können bei der Ausgabe die Werte der nachfolgenden Variablen aus der Argumentenliste eingefügt werden. Dazu muss die Zeichenkette für jeden auszugebenden Variablenwert ein Formatfeld enthalten. Die Formatfelder werden gemäß ihrer Reihenfolge den Variablen zugeordnet. Ein Formatfeld beginnt mit einem „`%`"-Zeichen und ist weiter unten in seiner Struktur erläutert.

Rückgabewert: Der Rückgabewert der **`printf`**-Funktion enthält die Anzahl der ausgegebenen Zeichen.

Struktur der Formatfelder der `printf`-Funktion:



Typangabe:

`d, i`	Ganzzahl mit Vorzeichen, dezimal
`o`	Ganzzahl ohne Vorzeichen, oktale Ausgabe
`x, X`	Ganzzahl ohne Vorzeichen, hexadezimale Ausgabe
`u`	Ganzzahl ohne Vorzeichen, dezimal
`c`	Ein einzelnes Zeichen
`s`	Zeichenkette (String)
`f`	Gleitkommazahl, in der Regel mit 6 Nachkommastellen gerundet
`g, G`	Gleitkommazahl in kürzester Darstellungsform, exponentiell oder fest
`e, E`	Gleitkommazahl in Exponentenform
`p`	Zeiger
`%%`	Ausgabe eines %-Zeichens

Beispiele:

Programmcode	Ausgabe
`int i=10;`	
`float f=123.;`	
`long int l=20;`	
`long double d=246.8;`	
`printf("%10d\n",i);`	`□□□□□□□□10`
`printf("%10.2f\n",f);`	`□□□□123.00`
`printf("%e\n",f);`	`1.230000e+002`
`printf("0x%08lx\n",l);`	`0x00000014`
`printf("%LE\n",d);`	`2.468000E+002`

Das Zeichen □ markiert ein Leerzeichen.

4.11 Eingabeanweisung mit `scanf`

Mit Hilfe der Funktion **`scanf`** können Werte für Variablen formatiert von der Tastatur eingelesen werden. Sie ist die zu **`printf`** analoge Eingabefunktion. Die allgemeine Syntax der Funktion lautet:

`[int-Variable=] scanf (`*Zeichenkette***`[,`***VarAdresse1***`, ... ,`***VarAdresseN***`]);`**

Optionale Elemente sind in eckigen Klammern eingeschlossen.

Argumente: Die als Argument übergebene Zeichenkette spezifiziert die Eingabeformate für die in der nachfolgenden Liste angeführten Variablen. Die Variablen dürfen nicht als Wert sondern müssen als **Adressen** an die **`scanf`**-Funktion übergeben werden. Damit ist es der **`scanf`**-Funktion möglich, die Inhalte der Variablen zu manipulieren.

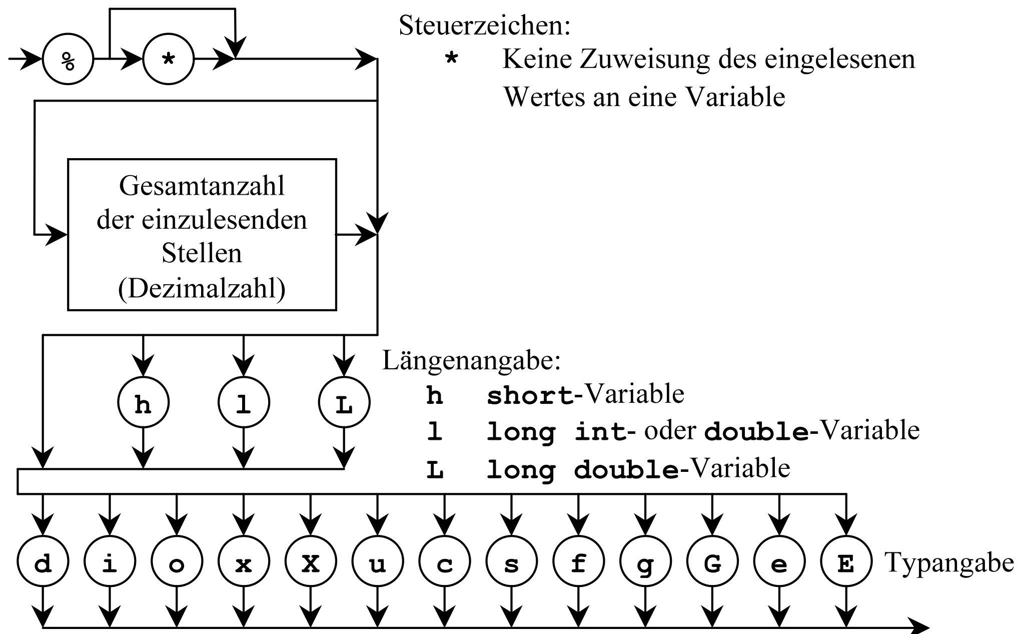
Die Adresse einer Variablen erhält man, indem man dem Variablennamen den **Adressoperator &** voranstellt:

Variablenadresse: &Variablenname

Jeder Variablen entspricht ein Formatfeld in der Zeichenkette. Zeichen, die zu keinem Formatfeld gehören (ohne %), müssen exakt im Eingabestrom enthalten sein und werden überlesen.

Rückgabewert: Als Rückgabewert wird die Anzahl der korrekt eingelesenen Variablenwerte geliefert. Endet der Eingabestrom ohne Einlesen einer Variable, wird EOF zurückgegeben.

Struktur der Formatfelder der `scanf`-Funktion:



Die Typangaben entsprechen der Tabelle der printf-Funktion.

Beispiel: Maximum von 2 Zahlen

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

main() {
   double a,b,max;
   printf("Geben Sie a und b ein");
   scanf("%lf %lf",&a,&b);
   max=((a+b) + fabs(a-b))/2.0;
   printf("Maximum von %f und %f lautet %f\n",
          a,b,max);
}
```

Warnungen:

- Ein Vergessen des &-Zeichens führt zu Fehlern und wird beim Übersetzen nicht erkannt!!!
- Die Werte der Eingabe können durch Leerzeichen, Zeilentrenner oder Tabs getrennt sein, aber nicht durch Kommata oder andere Zeichen.
- Anzahl und Reihenfolge der Formatangaben und der Adressvariablen müssen übereinstimmen!

4.12 Vergleichsoperatoren

`a<b`	kleiner
`a>b`	größer
`a<=b`	kleiner oder gleich
`a>=b`	größer oder gleich
`a==b`	gleich
`a!=b`	ungleich

Bei einem Vergleich findet vorher wieder eine Typumwandlung wie bei der arithmetischen Operation statt.

Jeder Vergleichsausdruck besitzt den Wahrheitswert 0, falls er falsch ist, und 1 bei Wahrheit:

```
a=2;
z=(a>1);  /* z hat den Wert 1, denn a>1 */
```

Die Vergleichsoperatoren binden schwächer als die arithmetischen Operatoren. Der Ausdruck:

```
5.0 * x+7 < y/4
```

entspricht beispielsweise dem folgenden Ausdruck mit überflüssigen Klammern:

```
((5.0 * x)+7) < (y/4)
```

Vergleichsausdrücke werden vor allem als Bedingungsausdrücke in **`if`**- oder **`for`**-Kontrollstrukturen (siehe weiter unten) eingesetzt:

```
#include <stdio.h>
#include <math.h>
main(){
   double x;
   scanf("%lf", &x);
   if (x>0)
      printf("ln(x)=%lf\n",log(x));
}
```

4.13 Kontrollstrukturen

4.13.1 Sequenz, Verbundanweisung

Die Sequenz (auch als Verbundanweisung bezeichnet) ist eine Zusammenfassung von Anweisungen, welche nacheinander durchgeführt werden. Eine Sequenz kann überall dort eingesetzt werden, wo eine einfache Anweisung gefordert wird.

```
{
   <Anweisung 1>;
   <Anweisung 2>;
   ...
   <Anweisung n>;
}
```

Anweisungen im Inneren der Sequenz können beispielsweise eine Zuweisung, ein Funktionsausdruck oder eine Kontrollstruktur sein.
Im Struktogramm kann für eine Verbundanweisung ein Block eingefügt werden oder einfach eine Sequenz von Verarbeitungsschritten:

Verarbeitungsschritt 1
Verarbeitungsschritt 2
Verarbeitungsschritt 3

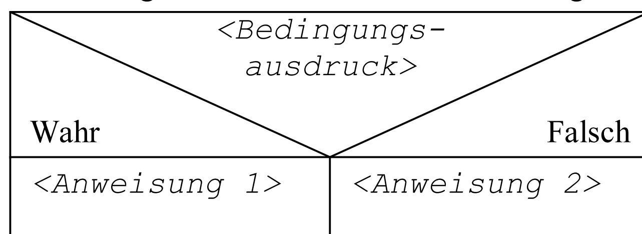
4.13.2 Einfache if-else Abfrage

Darstellung:

```
if ( <Bedingungsausdruck> )
   <Anweisung 1>;
else
   <Anweisung 2>;
```

Anweisung 1 wird nur dann ausgeführt, wenn die Bedingung erfüllt ist. Die Bedingung ist erfüllt, wenn der Bedingungsausdruck einen Wert $\neq 0$ liefert.

Liefert der Bedingungsausdruck den Wert 0, wird der **`else`**-Zweig und damit Anweisung 2 ausgeführt. Der **`else`**-Zweig kann fehlen.
Im Struktogramm wird die **`if-else`** Abfrage wie folgt dargestellt:

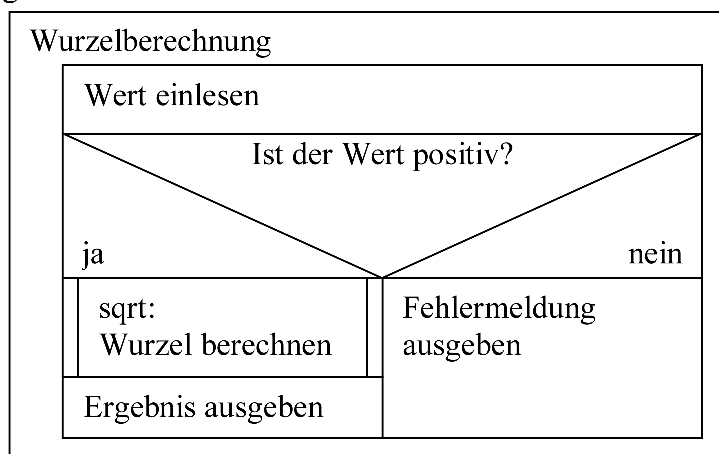


Durch Verwendung einer Sequenz statt einer einzelnen Anweisung können auch mehrere Anweisungen bei Eintreten der Bedingung und im **`else`**-Zweig ausgeführt werden:

```
if ( <Bedingungsausdruck> ){
   <Anweisung a>;
   <Anweisung b>;
   ...
} else {
   <Anweisung k>;
   <Anweisung l>;
   ...
}
```

Beispiel: Wurzelberechnung

Struktogramm:



Programm-Listing:

```
#include <stdio.h>
#include <math.h>
main () {
   double x;
   scanf ("%lf", &x);
   if (x>=0) {
      x=sqrt(x);
      printf ("%f", x);
   } else {
      printf ("x ist negativ \n");
   }
}
```

Geschachtelte **`if`**-Anweisungen:

Das folgende Beispiel zeigt eine Mehrdeutigkeit auf, die bei geschachtelten **`if`**-Anweisungen auftritt und durch eine Zusatzregel aufgelöst wird:

```
...
if (a==0)
   if (b>c)
      printf("%d\n",b);
   else
      printf("%d %d\n",a,c);
...
```

Da bei der **`if`**-Anweisung der **`else`**-Zweig fehlen darf, ist es im Beispiel nicht eindeutig, ob der gezeigte **`else`**-Zweig zur ersten oder zur zweiten **`if`**-Anweisung gehört. Gehört er zur ersten **`if`**-Anweisung, dann hat die zweite **`if`**-Anweisung keinen **`else`**-Zweig. Im zweiten Fall hat die erste **`if`**-Anweisung keinen **`else`**-Zweig.

Die Auflösung dieser Mehrdeutigkeit schafft eine Zusatzregel in C, die einen **`else`**-Zweig dem benachbarten **`if`** zuordnet.

Will man diese Regel umgehen, muss man explizit Klammern setzen und damit die gewünschte Zuordnung herstellen. Die Zuweisung des **`else`**-Zweigs zum ersten **`if`** erhält man beispielsweise wie folgt:

```
...
if (a==0) {
   if (b>c)
      printf("%d\n",b);
} else
   printf("%d %d\n",a,c);
...
```

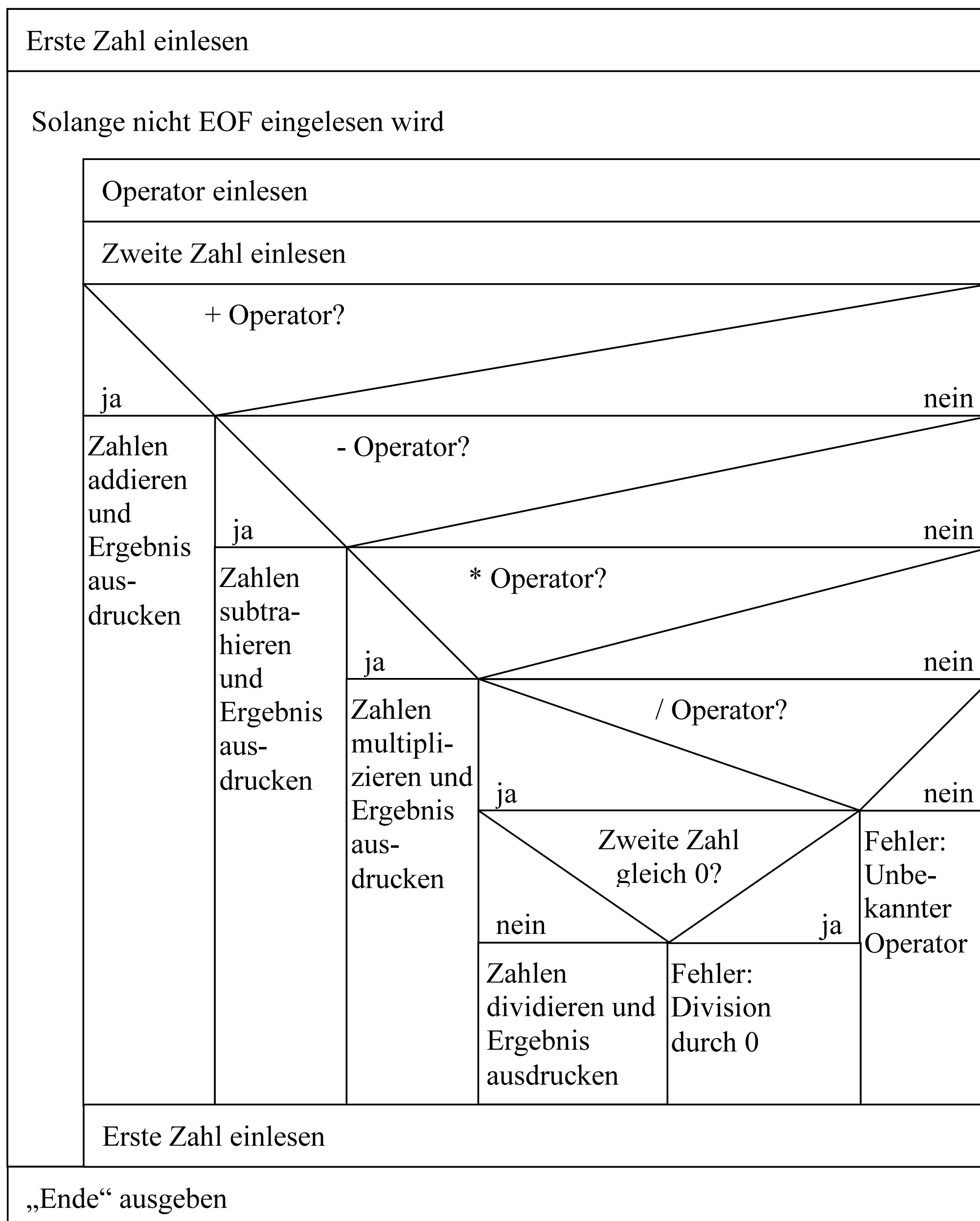
4.13.3 Mehrfachabfrage - else if

Darstellung:

```
if ( <Bedingung 1> )
   <Anweisung 1>;
else if ( <Bedingung 2> )
   <Anweisung 2>;      /* ... */
else if ( <Bedingung n> )
   <Anweisung n>;
else <Anweisung n+1>;
```

Nur die erste Anweisung, deren Bedingung erfüllt ist, wird ausgeführt. Ist keine Bedingung erfüllt, so wird, falls vorhanden, die letzte `else`-Anweisung (n+1) ausgeführt.
Als Anweisung kann auch eine Verbundanweisung mit einer Sequenz von Anweisungen in geschweiften Klammern ausgeführt werden.

Beispiel: Taschenrechner



Programm-Listing:

```
main()
{
   float x,y;
   char operator;

   printf("\nGib eine Zahl ein >");


   while (EOF!=scanf("%f",&x)){
      printf("\nGib einen Operator ein (+/-) >");
      getchar();
      operator=getchar();
      printf("\nGib noch eine Zahl ein >");
      scanf("%f",&y);
      if (operator=='+')
         printf("\ndas Ergebnis: %f",x+y);
      else if (operator=='-')
         printf("\n das Ergebnis: %f",x-y);
      else printf("ungueltige Eingabe :%c",operator);
      printf("\nGib eine Zahl ein >");
      }
   printf("\nEnde\n\n");
}
```

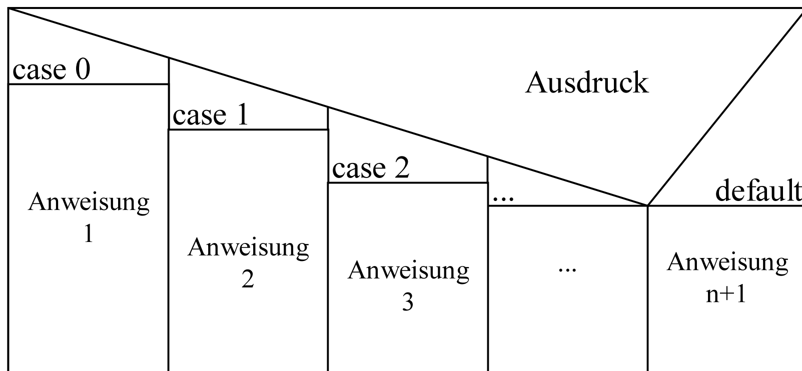
4.13.4 Mehrfachabfrage - switch

Darstellung:

```
switch ( <Ausdruck> ) {
   case <konstanter Ausdruck 1>:
      <Anweisung 1>;
      break;
   case <konstanter Ausdruck 2>:
      <Anweisung 2>;
      break;
   ...
   case <konstanter Ausdruck n>:
      <Anweisung n>;
      break;
   default:
      <Anweisung n+1>;
}
```

Zunächst wird der Wert des Ausdrucks in der **`switch`**-Zeile errechnet. Dieser Wert wird mit den Werten *`<konstanter Ausdruck i>`* der **`case`**-Zeilen von oben beginnend verglichen. Jeder konstante **`case`**-Ausdruck darf nur einmal vorkommen. Stimmt der Wert von *`<Ausdruck>`* mit einem konstanten **`case`**-Ausdruck überein, werden alle dieser **`case`**-Zeile nachfolgenden Anweisungen bis zur **`break`**-Anweisung oder bis zum Ende des **`switch`**-Blocks ausgeführt. Die **`break`**-Anweisung sorgt dafür, dass der **`switch`**-Block verlassen wird. Ist keine der Bedingungen erfüllt, so werden, falls vorhanden, die Anweisungen hinter der **`default`**-Zeile ausgeführt.

Die **`switch`**-Anweisung wird durch das nachfolgende Struktogramm visualisiert:



Beispiel:

```
#include <stdio.h>
main(){
   /* Vereinbarung der Variablen */
   char a;
   /* Programmcode */
   a=getchar(); /* Zeichen einlesen */
   switch (a) {
      case 'a':
      case 'A':
         printf("\n a oder A eingegeben. \n");
         break;
      case 'b':
      case 'B':
         printf("\n b oder B eingegeben. \n");
         break;
      default:
         printf("\n Falsche Eingabe \n");
   }
}
```

4.13.5 while - Schleife

Darstellung:

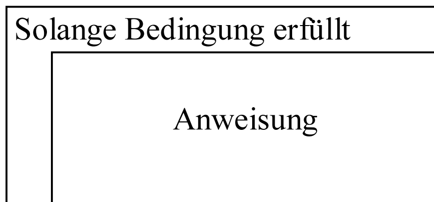
```
while ( <Bedingung> )
   <Anweisung>;
```

Die **`while`**-Schleife ist eine kopfgesteuerte Schleife. Die enthaltene Anweisung wird ausgeführt, solange die Bedingung erfüllt ist. Somit muss die Anweisung irgendwie Einfluss auf die Bedingung nehmen, sonst wird die Schleife unendlich oft ausgeführt.

Natürlich kann als Anweisung innerhalb der **`while`**-Schleife kann auch eine Verbundanweisung verwendet werden:

```
while ( <Bedingung> ) {
   <Anweisung 1>;
   <Anweisung 2>;
   <Anweisung n>;
}
```

Das Struktogramm für die kopfgesteuerte Schleife hat die folgende Form:



Beispiel: Bestimmung der höchsten Dezimalziffer einer natürlichen Zahl

```
#include <stdio.h>
main() {
   int Zahl;
   scanf("%d", &Zahl);
   while (Zahl>=10)
      Zahl=Zahl/10;
   printf("Höchste Dezimalziffer: %d\n", Zahl);
}
```

4.13.6 do-while - Schleife

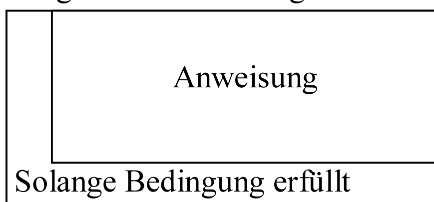
Darstellung:

```
do
   <Anweisung>;
while ( <Bedingung> );
```

Die **do-while** - Schleife ist ein fußgesteuerte Schleife. Zunächst wird die Anweisung einmal ausgeführt. Solange die Bedingung erfüllt ist, wird die Anweisung erneut ausgeführt. Als Anweisung ist auch eine Verbundanweisung möglich:

```
do {
   <Anweisung 1>;
   <Anweisung 2>;
   <Anweisung n>;
} while ( <Bedingung> );
```

Das Struktogramm für die fußgesteuerte Schleife hat die folgende Form:



4.13.7 for - Schleife

Darstellung:

```
for ( <Ausdruck 1>; <Bedingung>; <Ausdruck 2> )
   <Anweisung>;
```

Bei der Ausführung der **for**-Schleife werden die folgenden Schritte abgearbeitet:

1. Falls vorhanden wird der *<Ausdruck 1>* ausgewertet.
2. Die *<Bedingung>* wird ausgewertet.
   Bei erfüllter (oder fehlender) Bedingung wird die *<Anweisung>* ausgeführt.
   Ist die Bedingung nicht erfüllt, wird die Schleife beendet.
3. *<Ausdruck 2>* wird ausgewertet.
4. Fortsetzung mit Punkt 2.

Beispiel: Ausdruck aller Zahlen von 0 bis 99

```
#include <stdio.h>
main () {
    int i;
    for (i=0; i<100; i=i+1)
        printf ("%d\n", i);
}
```

Die **for**-Schleife ist gleichbedeutend mit folgendem Code-Fragment:

```
<Ausdruck 1>;
while ( <Bedingung> ){
   <Anweisung>;
   <Ausdruck 2>
}
```

Statt einer einzelnen Anweisung kann in der for-Schleife auch eine eine Verbundanweisung angegeben werden:

```
for ( <Ausdruck 1>; <Bedingung>; <Ausdruck 2> ) {
   <Anweisung 1>;
   <Anweisung 2>;
   <Anweisung n>;
}
```

Für die **for**-Schleife gibt es kein spezielles Struktogramm. Es kann das Struktogramm der **while**-Schleife verwendet werden, wenn man als Kopfbedingung die Ausdrücke und die Bedingung der **for**-Schleife einträgt.

Bei der **for**-Schleife dürfen `<Ausdruck 1>`, `<Bedingung>` und/oder `<Ausdruck 2>` fehlen; die Semikolons müssen aber geschrieben werden. Nachfolgendes Programmfragment realisiert beispielsweise eine Endlosschleife:

```
...
for (;;)
   <Anweisung>;  /* Endlosschleife */
...
```

4.14 Steuerung von Schleifen durch `break` und `continue`

Bisher wurden die Schleifen allein durch die Kopf- oder Fußbedingungen gesteuert, der Anweisungsteil wurde als Ergebnis der Bedingung unbedingt abgearbeitet.
Zur Fortschaltung oder Beenden einer Schleife aus dem Anweisungsteil heraus dienen die **`break`**- und die **`continue`**-Anweisung.

Die **`break`**-Anweisung wurde bereits im Zusammenhang mit der **`switch`**-Anweisung behandelt. Sie führte dort zu einem sofortigen Beenden der **`switch`**-Anweisung. Gleiches gilt bei Schleifen. Eine **`break`**-Anweisung führt zum sofortigen Beenden der aktuellen Schleife, ohne dass Kopf- oder Fußbedingungen geprüft werden. Die Programmausführung wird mit den Anweisungen hinter der Schleife fortgesetzt.

Beispiel: Kopierprogramm (Version 2)

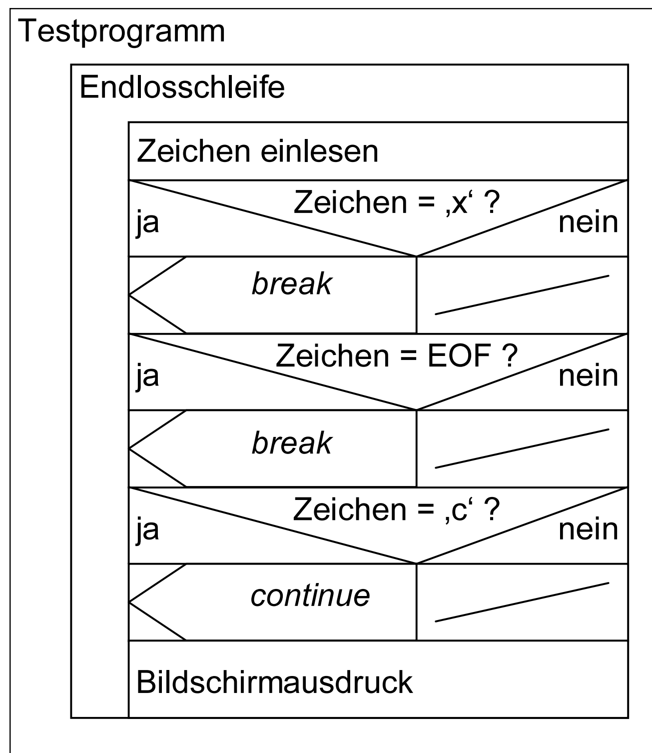
```
#include <stdio.h>
main(){
   int c;
   while(1) {
      c=getchar();
      if (EOF==c)
         break;
      else
         putchar(c);
   }
}
```

Die **`continue`**-Anweisung bewirkt, dass der aktuelle Durchlauf des Anweisungsteils einer Schleife beendet wird. Bei **`for`**-Schleifen wird dann der *<Ausdruck 2>* ausgewertet und bei allen Schleifentypen die *<Bedingung>* der Schleife geprüft. Ist die Bedingung erfüllt, wird ein neuer Durchlauf des Anweisungsteils gestartet.
Im Struktogramm ist die Darstellung der break- und continue-Anweisung nicht fest vorgegeben. Häufig wird das folgende Symbol für beide Anweisungen verwendet, die Unterscheidung erfolgt durch Benamung:

Beispiel:

```
#include <stdio.h>
main(){
   int c;
   for(;;) {
      c=getchar();
      if (c=='x') break;
      if (c==EOF) break;
      if (c=='n') continue;
      printf("Weder x noch c gedrueckt\n");
   }
}
```

Zugehöriges Struktogramm:



4.15 Logische Operatoren

Es gibt folgende logische Operatoren:

`!`	Nicht
`&&`	UND
`||`	ODER

Sie erzeugen aus Wahrheitswerten neue Wahrheitswerte. Bei Wahrheit liefern sie den Wert 1, sonst 0. Die Operanden können Zahlenwerte beliebigen Typs sein (also auch **`float`**, **`double`**). Ein Wert ≠ 0 steht für Wahrheit, ein Wert = 0 für Unwahrheit.

Die logischen Operatoren **`&&`** und **`||`** binden schwächer als die bereits bekannten Vergleichsoperatoren und arithmetischen Operatoren. Der Negationsoperator **`!`** bindet aber stärker. Der **`&&`**-Operator bindet stärker als der **`||`**-Operator.

Die Auswertung erfolgt von links nach rechts und wird abgebrochen, sobald das Ergebnis feststeht.

```
<Ausdruck 1> || <Ausdruck 2>
```

Ausdruck 2 wird nur dann ausgewertet, falls der Ausdruck 1 falsch ist.

```
<Ausdruck 1> && <Ausdruck 2>
```

Ausdruck 2 wird nur dann ausgewertet, falls der Ausdruck 1 wahr ist.

Beispiele:

a) `4.5 < x && x < 9.0`

liefert den Wahrheitswert 1, falls 4.5 < x < 9.0 ist.

b) `!-7.45e+4` hat den Wahrheitswert 0

c) `!a == 0` wahr, wenn a ≠ 0; gleichbedeutend mit `(!a)==0`

d) `x && y && z` ist gleichbedeutend zu `(x && y) && z`

e) `x1 || x2 && ! x3 && x4 || x5`

ist gleichbedeutend mit:

`(x1 || ((x2 && (!x3)) && x4) || x5`

f) `(c >= '1' && c <= '9') || (c >= 'A' && c <= 'Z')`

Wahr, falls c eine Ziffer oder ein großer Buchstabe zwischen A und Z ist.

g) `b > c && d == a || c <= a`

ist gleichbedeutend mit:

`((b > c) && (d == a)) || ( c<= a)`

4.16 Inkrement und Dekrement

Die Addition und Subtraktion von 1 zu bzw. von einer Variablen ist eine sehr häufige Operation. Deshalb gibt es für diesen Zweck zwei spezielle Operatoren

`++` und `--`.

Beispiele:

`a++;` entspricht `a=a+1;`

`a--;` entspricht `a=a-1;`

Der Inkrement- und Dekrement-Operator kann sowohl vor als auch nach der Variablen stehen:

Präfix-Notation: `++a` bzw. `--a`

Postfix-Notation: `a++` bzw. `a--`

In der Präfixnotation wird a inkrementiert bzw dekrementiert bevor die Variable genutzt wird. In der Postfixnotation erfolgt die Modifikation der Variablen nach deren Nutzung.

Beispiel:

a) `a=5*(b++);` ist gleichbedeutend mit `a=5*b; b=b+1;`

b) `a=5*(++b);` ist gleichbedeutend mit `b=b+1; a=5*b;`

Ist z.B. b=1, so errechnet sich b in beiden Fällen zu 2. Die Variable a wird im ersten Fall zu 5 gesetzt, im zweiten Fall zu 10.

Inkrement und Dekrement können auf jeden Variablentyp angewendet werden. Sie binden stärker als die arithmetischen Operatoren.

Häufige Verwendung in Zählschleifen:

```
for(i=0;i<N;i++)
    ...
```

Vorsicht bei mehrfacher Anwendung dieser Operatoren:

```
a=b++  +  b++; /* Compiler-abhängig*/
```

Das Ergebnis ist Compiler-abhängig, da unklar ist, ob vor der zweiten Verwendung von b die erste Inkrementoperation schon durchgeführt wurde.

4.17 Zuweisungsoperatoren

Neben der einfachen Zuweisung kann eine Zuweisung eine mit einer arithmetischen Operation verknüpft werden. Die Beispiele zeigen, wie der Operand auf der linken Seite der Zuweisung in die arithmetische Berechnung einbezogen wird.

`=`	Einfache Zuweisung
`+=`	Addition und Zuweisung
`-=`	Subtraktion und Zuweisung
`*=`	Multiplikation und Zuweisung
`/=`	Division und Zuweisung
`%=`	Modulo und Zuweisung

Beispiele:

```
x *=4;                    x=x*4;
y-=1;                     y=y-1;
w/=23;                    w=w/23;
jj*=xx++  + ++yy;         jj=jj*(xx++  +  ++yy);
```

4.18 Shiftoperatoren

`<<`	Linksverschiebung der Bitkette
`>>`	Rechtsverschiebung der Bitkette

Beispiel:

```
x << 3 /* Entspricht x*8 */
y >> 4 /* Entspricht x/16 */
1 << 4 /* Entspricht 2 hoch 4 */
```

Linksverschiebung:

Der Binärwert des linken Operanden (int-Typ: char, short, int, long) wird soviele Bitstellen nach links verschoben, wie der rechte Operand (immer positiv!) angibt. Die unteren Bitstellen werden in der Regel mit Nullen aufgefüllt.

Rechtsverschiebung:

Der Binärwert des linken Operanden (int-Typ: char, short, int, long) wird soviele Bitstellen nach rechts verschoben, wie der rechte Operand (immer positiv!) angibt. Die oberen Bitstellen werden in der Regel mit Nullen aufgefüllt.

Achtung: Wird ein Typ mit Vorzeichen nach rechts verschoben, so wird auf manchen Systemen das Vorzeichenbit, also 1, auf anderen Systemen 0 nachgeschoben.

4.19 Bit-Operatoren

`&`	bitweises UND
`|`	bitweises (inkl.) ODER
`^`	bitweises Exklusiv-ODER (XOR)
`~`	1-Komplement (Vertauschen 0-1/Inversion)

Bitoperationen werden genutzt, um auf eine Auswahl von Bits innerhalb einer Variablen zuzugreifen.

Beispiele:

```
short int x,y; /* 16 Bit Werte*/
y=x | 0xF000;
x=x & 077;
x=x & ~077; /* gleichbedeutend mit   x=x & 0177700 */
x=x & (1<<5) /* maskiert Bit 5 von x. */
y=x ^ ((1<<3)|(1<<5)|(1<<7)); /* invertiert Bits 3, 5 und 7 */
```

Der Unterschied zwischen **`x&&y`** und **`x&y`** ist zu beachten. Für x=1, y=2 ist **`x&&y=1`** (wahr) und **`x&y=0`**.

4.20 Fragezeichenoperator

Syntax:

```
( <Bedingung> ) ? <Ausdruck1> : <Ausdruck2>
```

Ist die *<Bedingung>* erfüllt, liefert der Operator den Wert von *<Ausdruck1>*, sonst von *<Ausdruck2>*.

Beispiel:

```
z=(a>b) ? a : b ;  /* Maximum von a und b */
```

dies entspricht dem folgenden Code:

```
if (a>b) z=a; else z=b;
```

Beispiel:

```
z=(a>0) ? a:-a ;  /*Absolutbetrag von a  */
```

dies entspricht dem folgenden Code:

```
if (a>0) z=a; else z=-a;
```

Man beachte, dass vor der Auswertung Typanpassungen wie bei arithmetischen Verknüpfungen stattfinden. Der Ausdruck:

```
a>0?1:2.0
```

liefert für a>0 den Wert 1 mit Datentyp **`double`**.

Wie sonstige Ausdrücke kann man bedingte Ausdrücke beliebig tief schachteln:

```
x>=0. ? x==0. ? 0:1:-1; /* Vorzeichen: sign(x) */
```

ist gleichbedeutend mit

```
x>=0. ? (x= =0.? 0:1):-1
```

4.21 Bindungsstärken von Operatoren

Nachfolgende Tabelle gibt eine vollständige Übersicht über die Operatoren von C.

<table>
<tr><th>Priorität</th><th>Assoziativität</th><th>Operatoren</th><th>Bezeichnung</th></tr>
<tr><td rowspan="3">1</td><td rowspan="3">links</td><td><code>()</code></td><td>Funktionsaufruf</td></tr>
<tr><td><code>[]</code></td><td>Feldzugriff</td></tr>
<tr><td><code>-> .</code></td><td>Komponentenselektion</td></tr>
<tr><td rowspan="5">2</td><td rowspan="5">rechts</td><td><code>++ --</code></td><td>Inkrement, Decrement</td></tr>
<tr><td><code>! ~</code></td><td>Negation (Logisch, Bitweise)</td></tr>
<tr><td><code>+ -</code></td><td>Vorzeichen (Unär)</td></tr>
<tr><td><code>* &</code></td><td>Dereferenzierung, Adresse</td></tr>
<tr><td><code>(type)</code></td><td>Typkonversion</td></tr>
<tr><td>3</td><td>links</td><td><code>* / %</code></td><td>Arithmetische Operatoren</td></tr>
<tr><td>4</td><td>links</td><td><code>+ -</code></td><td>Arithmetische Operatoren</td></tr>
<tr><td>5</td><td>links</td><td><code><< >></code></td><td>Schiebeoperatoren</td></tr>
<tr><td>6</td><td>links</td><td><code>< <= > >=</code></td><td>Vergleichsoperatoren</td></tr>
<tr><td>7</td><td>links</td><td><code>== !=</code></td><td>Vergleichsoperatoren</td></tr>
<tr><td>8</td><td>links</td><td><code>&</code></td><td>Bitweises UND</td></tr>
<tr><td>9</td><td>links</td><td><code>^</code></td><td>Bitweises Exklusiv-Oder</td></tr>
<tr><td>10</td><td>links</td><td><code>|</code></td><td>Bitweises Oder</td></tr>
<tr><td>11</td><td>links</td><td><code>&&</code></td><td>logische Und</td></tr>
<tr><td>12</td><td>links</td><td><code>||</code></td><td>logisches Oder</td></tr>
<tr><td>13</td><td>rechts</td><td><code>? :</code></td><td>Fragezeichenoperator</td></tr>
<tr><td>14</td><td>rechts</td><td><code>= += -= *= /= %=
^= &= |= <<= >>=</code></td><td>Zuweisungsoperatoren</td></tr>
<tr><td>15</td><td>links</td><td><code>,</code></td><td>Komma-Operator</td></tr>
</table>

In der Tabelle sind die Operatoren gemäß ihrer Priorität angeordnet. Die Priorität gibt an, in welcher Reihenfolge die Operatoren in einem Ausdruck abgearbeitet werden. Die Operatoren verknüpfen in Reihenfolge absteigender Priorität die Operanden eines Ausdrucks.
Innerhalb einer Priorität bestimmt die Assoziativität die Abarbeitungsreihenfolge der Operatoren. Bei Rechts-Assozivität werden die Operatoren gleicher Prioritätsstufe von rechts nach links abgearbeitet, Bei Links-Assozivität entsprechend von links nach rechts.

Beispiele:

`!++a` ist gleichbedeutend mit `!(++a)`

Erhöhung von a um 1, anschließend logische Verneinung von ++a

`-a++` gleichbedeutend mit `-(a++)`

(Liefert als Wert -a, Anschließend erfolgt eine Erhöhung von a um 1)

`(a<0)?(b>4)?x:y:z` gleichbedeutend `(a<0)?((b>4)?x:y):z`

Achtung:
In C ist nicht festgelegt, in welcher Reihenfolge die Argumente einer Funktion ausgewertet werden. Die Anweisung

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

ist nicht eindeutig. Besser:

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

Im Zweifelsfall und zur besseren Übersichtlichkeit: Klammern setzen!!!

4.22 Felder

Gleichartige, logische zusammenhängende Objekte können zu einem Feldern (Array, Vektor) zusammengefasst werden.
Vorteil:

- Für gleichartige Objekte wird eine einzelne, gemeinsame Variable verwendet.
- Die einzelnen Objekte sind gemeinsam ansprechbar.

Syntax:

Vereinbarung/Definition:

<Typ> <Name>`[`*<N>*`];` (N: Integer-Wert)

Aufruf:

<Name>`[`*<Index>*`]` (mit 0<=Index<=N-1)

Es werden N Speicherplätze des angegebenen Typs reserviert, die mit den Bezeichnern *<Name>*`[0]`, ..., *<Name>*`[`*<N-1>*`]` angesprochen werden können.

Beispiel:

Es wird ein Vektor mit Namen **`vekt`** bestehend aus 5 **`int`** Zahlen vereinbart, d.h. es wird ein zusammenhängender Speicherplatz für 5 **`int`**-Zahlen reserviert:

```
int vekt[5];
```

Die Elemente 0 und 1 des Vektors werden auf die Werte 6 und 4 gesetzt:

```
vekt[0]=6;
vekt[1]=4;
```

Der Vektor hat nun das folgende Erscheinungsbild:

`vekt`:	**`vekt[0]`**	**`vekt[1]`**	**`vekt[2]`**	**`vekt[3]`**	**`vekt[4]`**
	6	4	?	?	?

Bemerkungen:

- Das erste Feldelement von Vektoren hat immer den Index 0 das letzte hat den Index N-1.
- In C erfolgt beim Zugriff auf Vektorelemente **keine** Prüfung auf mögliche Bereichsüberschreitungen. In Anlehnung an obiges Beispiel wird beispielsweise mit:
  ```
  vekt[5]=7;
  ```
  der Speicherplatz hinter dem letzten Element **`vekt[4]`** des Vektors überschrieben und damit zerstört. Dies führt zu einem Fehler, der erst zur Laufzeit des Programms (hoffentlich) erkannt wird (Laufzeitfehler).

Initialisierung von Vektoren:

Die Elemente eines Vektoren können schon bei der Definition des Vektors initialisiert werden. Dazu wird eine in geschweiften Klammern eingeschlossene Liste von initialen Werten mit einem Gleichheitszeichen an die Definition angehängt:

```
<Typ> <Name>[<N>]={<Wert0>,<Wert1>,...};
```

Die Liste kann weniger Initialisierungswerte enthalten als der Vektor Elemente besitzt, dann werden die hinteren Elemente des Vektors, für die keine initialen Werte angegeben sind, zu 0 initialisiert.
Bei vorhandener Initialisierungsliste braucht weiterhin die Größe des Vektors (Dimension) nicht spezifiziert werden, dann wird die Anzahl der Elemente aus der Länge der Initialisierungsliste ermittelt:

```
<Typ> <Name>[]={<Wert0>,<Wert1>,...,<WertN>};
```

Beispiele:

Vollständige Initialisierung eines Vektors:

```
int tage[6]={31,28,31,30,27,31};
```

Anzahl der Initialisierungswerte ergibt die Dimension des Vektors:

```
int tage[]={31,28,31,30,27,31};
```

Unvollständige Initialisierungsliste für einen Vektor. Die restlichen Elemente werden mit 0 initialisiert.

```
int n[5]={1,2,3};  /* n[3]=0, n[4]=0 */
```

Folgendes Programm liest einen Vektor der Dimension 5 ein und gibt den Betrag (Länge) des Vektors aus.

```
#include <stdio.h>

#include <math.h>

main() {
   int i;

   float n[5],sum=0,skprod;

   for(i=0;i<5;i++) {
      scanf("%f",&(n[i]));
      sum=sum+pow(n[i],2);
   }
   skprod=sqrt(sum);
   printf("Betrag: %f ",skprod);
}
```

4.23 Zeichenketten

In C gibt es keine eigenen Datentyp für Zeichenketten (String). Zum Speichern einer Zeichenkette wird ein Vektor vom Typ char vereinbart. Im letzten Element dieses Zeichenketten-Vektors wird das **Endezeichen '\0'** abgelegt.

Beispiel:

Die folgenden alternativen Definitionen vereinbaren einen Zeichenketten-Vektor, der die Zeichenkette „hallo“ speichert. Da neben den Buchstaben auch das Endezeichen abgelegt werden muss, hat der Vektor die Länge 6.

```
char text[6]={'h','a','l','l','o','\0'};
```

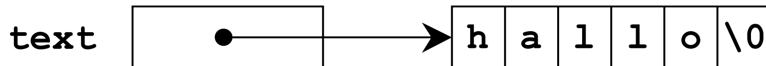
oder

```
char text[]="hallo";
```

oder

```
char *text="hallo";
```

Jede dieser alternativen Definitionen legt für den Vektor **text** einen Zeiger fest, der auf den Speicherbereich zeigt, wo die Elemente des Vektors abgelegt sind:



Nahezu alle C-Programme, die Zeichenketten verarbeiten, erwarten als Abschluss einer Zeichenkette das Endezeichen `'\0'`. Jeder Programmierer muss selbst dafür sorgen, dass seine Zeichenketten mit `'\0'` abschließen!
Werden Zeichenketten unbekannter Länge verarbeitet, so muss ein **char**-Vektor mit genügend vielen Elementen angelegt werden, um alle möglichen Zeichenketten darin abzulegen. Dieser Vektor wird mit den Zeichen der aktuellen Zeichenkette gefüllt. Das Endezeichen wird hinter das letzte gültige Zeichen der aktuellen Zeichenkette gesetzt, die hinterliegenden Zeichen können beliebige Werte enthalten.
String-Konstanten, d.h Zeichenketten in Anführungszeichen (z.B. **"WESTERKAMP"**), sind in C ebenfalls Zeichenketten-Vektoren. Bei Vereinbarung einer String-Konstante wird ein Feld ein Zeiger auf die erste Feldkomponente angelegt. Damit ist der Zugriff auf einzelne Elemente mit den bekannten Operatoren möglich:

`"WESTERKAMP"[3]`	entspricht	`'T'`
`"LANG"[4]`	entspricht	`'\0'`
`*"GERVENS"`	entspricht	`'G'`
`*("Lang"+2)`	entspricht	`'n'`

Beispiel Kopierfunktion, Vektorvariante

```
void strcpy(char *quelle, char *ziel) {
   int i=0;
   while((ziel[i]=quelle[i])!='\0')
      i++;
}
```

Beispiel Kopierfunktion, Zeigervariante

```
void strcpy(char *quelle, char *ziel) {
   while(*ziel++=*quelle++) ;
}
```

Verfügbare Zeichenkettenfunktionen:
In der Headerdatei **string.h** sind Deklarationen für viele Zeichenkettenfunktionen enthalten:

`int strlen(char *s);`	Ermittelt Länge von s (ohne \0)
`int strcmp(char *s, char *t);`	Vergleich zweier Strings (=0: s gleich t, <0: s vor t, >0: s hinter t)
`int strncmp(char *s, char *t, int n);`	Vergleich wie strcmp aber höchstens n Zeichen
`char *strcpy(char *s,char *t);`	kopiert t nach s inkl \0, liefert s
`char *strncpy(char *s,char *t, int n);`	kopiert höchstens n Zeichen
`char *strcat(char *s,char *t);`	kopiert t an das Ende von s
`char *strchr(char *s,int t);`	Sucht das Zeichen t in s, liefert Zeiger in s
`char *strstr(char *s, char *t);`	sucht t als String in s (Substring)

Bei Verwendung dieser Funktionen bindet der Linker den zugehörigen Code für die Funktionen in das ausführbare Programm ein.
Man beachte, dass diese Funktionen keine Speicherverwaltung übernehmen. Der Speicherbereich für vorkommende Zeichenketten muss also vorher bereitgestellt werden!

Beispiel: Einlesen einer Zeichenkette

```
char z[100]; /* maximal 100 Zeichen vorsehen */
/* Zeile einlesen bis zum ersten \n */
gets(z);
```

oder:

```
char z[100]; /* maximal 100 Zeichen vorsehen */
/* wortweise bis zum 1. white space einlesen */
scanf("%s",z);
```

Beide Funktionen fügen ein \0 an.

Beispiel: Ausgabe einer Zeichenkette

```
char z[100]="Hallo";
/* Ausgabe bis zum ersten \0, es wird ein \n angehängt: */
puts(z);
printf("%s",z);
/* Linksbündig, m Mindestbreite, n Anzahl auszudruckender Zeichen: */
printf("%-m.ns",z);
/* Es werden i Zeichen ausgedruckt: */
printf("%.*s",i,s);
```

Beispiel: Text verändern

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

main() {
   char z[100], umkehr[100], up[100], low[100];
   int i, laenge;

   printf("Text eingeben (max. Laenge 100)\n");
   gets(z);
   laenge=strlen(z);

   /* Grossbuchstaben */
   for(i=0; i<laenge; i++) {
      up[i]=toupper(z[i]);
   }
   up[laenge]='\0';
   printf("Gross:\t%s\n", up);

   /* Kleinbuchstaben */
   for(i=0; i<laenge; i++) {
      low[i]=tolower(z[i]);
   }
   low[laenge]='\0';
   printf("Klein:\t%s\n", low);

   /* Rueckwaerts */
   for(i=0; i<laenge; i++) {
     umkehr[i]=z[laenge-1-i];
   }
   umkehr[laenge]='\0';
   printf("Umkehrung:\t%s\n", umkehr);
}
```

4.24 Tipps und Tricks zum Lösen der Aufgaben

a) Arbeiten unter Open Windows mit einem Texteditor-Fenster zum Schreiben des C-Programms und einem Komando-Fenster zum Compilieren.

b) Die letzten Befehle kann man sich mit **`h`** (history) anzeigen lassen. Mit !(Nr) kann der Befehl mit der entsprechenden Nummer wieder ausgeführt werden, ohne dass er wieder ganz eingegeben werden muss.

c) Eine weitere Eingabehilfe ist in der Verwendung der Cut, Copy und Paste Tasten gegeben. Erst einen Text markieren, dann in den Buffer kopieren und dann an gewünschte Stelle wieder einfügen.

d) Beim Compilieren kann man den Namen der zu erzeugenden Maschinencode-Datei angeben durch:

```
cc auf1.c -o auf1.exe
```

e) Unter UNIX gibt es das Kommando
`cb` (c-beautifier)
`cb aufg1.c` liest die die Datei **`aufg1.c`** und schreibt es in gut lesbarer C-Struktur auf den Bildschim

f) Unter UNIX kann man die Standardein- und -ausgabe in eine beliebige Datei umleiten:

```
a.out > file.aus
a.out < file.ein
file.ein > a.out > file.aus
```

`a.out` liest die Eingabedaten aus **`file.ein`** und schreibt die Ausgabedaten in die Datei **`file.aus`**. Dies gilt auch für die meisten UNIX-Kommandos:
z. B.: **`cb aufg1.c > schoen1.c`**

g) Man kann auch zwei Programme gleichzeitig starten und die Ausgabe des ersten in die Eingabe des zweiten lenken (Pipe oder Pipeline):

```
auf1.exe | auf2.exe
```

Analog bei mehreren Programmen:

```
auf1.exe | auf2.exe | auf3.exe | ...
auf1.exe < File.ein| auf2.exe | auf3.exe > File.out
```

h) Im Texteditor besteht im Menüpunkt **`View`** die Möglichkeit, bestimmte Zeilennummern anzusprechen, in denen sich möglicherweise ein Fehler befindet.

i) Die Beschreibung von UNIX-Kommandos und C-Funktionen kann man sich mit **`man`** (Manual) anzeigen lassen, z. B.: **`man cb`**

j) Unter Open Windows findet man die C- und UNIX-Handbücher im Open Windows Menu Answerbook.