Programmieren in C

-- ALLE Programmiersprachen sind HÄSSLICH --

Die einfachste Programmstruktur:

```
main ()
{

-- was zu tun ist ---

}
```

Vorgeordnete Definitionen:

```
# include <Header-Datei>  -- (.h)

#define Name Wert

-- Deklaration der globalen Variablen --

-- Funktionen --

main ()
{

-- Deklaration der Variablen --

-- was zu tun ist ---

}
```

Header-Dateien für Borland Turbo-C (mit Port-EA):

```
#include <stdio.h>, <conio.h>, <dos.h>, <string.h>
```

Datentypen:

Wir brauchen nur int und char.

Deklaration: erst der Datentyp, dann der Variablenname. Semikolon am Ende.

Anfangswertzuweisung: im Anschluß an den Namen. Beispiel:

```
int Temperatur = 20;
```

Deklaration von Feldern (Arrays):

```
Datentyp  Variablenname [Anzahl];      -- der erste Index ist stets 0.
                                       -- Indexbereich: 0...Anzahl-1
```

Zugriff auf eine Variable im Array:

```
Variablenname [Index]
```

Mehrdimensionale Arrays:

```
Datentyp  Variablenname [1. Anzahl] [2. Anzahl] usw.;
```

Zeichenketten als Datentyp: ham wa nich...

Ausweg: ein Array vom Typ char definieren. Muß so groß sein, daß es auch die längste Zeichenkette aufnehmen kann.

Automatisches Füllen eines solchen Arrays mit einer Zeichenkette:

```
strcpy (Array, Zeichenkettenkonstante);
```

Automatisches Füllen eines solchen Arrays durch Eingabe:

```
gets (Array);
```

Aneinanderhängen von Zeichenketten:

```
strcat (Array, Zeichenkettenkonstante);
```

Die Zeichenkettenkonstante wird an die im Array vorhandene Zeichenkette angehängt, so daß das Array wieder nur eine einzige Zeichenkette enthält.

ACHTUNG: es gibt keine Längenprüfung. Ggf. wird der Speicherplatz hinter dem Array gnadenlos überschrieben!

Verarbeiten von Arrays mit Zeichenketten: in Schleifen (zeichenweise).

Wie lang ist eine Zeichenkette?
Zeichenketten werden mit einem Zeichen 00H abgeschlossen (`\0´).

Längenermittlung:

```
strlen (Array);  -- das abschließende Nullzeichen wird nicht mitgezählt.
```

Konstanten:

1. ganze Zahlen:

a) dezimal: als gewöhnliche Dezimazahl. Z. B. 4711. Achtung: keine führenden Nullen. Also NICHT 0815.

b) oktal (1 Ziffernstelle = 3 Bits; Wertebereich 0...7): mit vorangestellter 0. Beispiel: 0715 (= dezimal 461).

c) hexaezimal (1 Ziffernstelle = 4 Bits; Wertebereich 0...9, A...F): mit vorangestelltem 0x. Beispiel: 0x715 (= dezimal 1813).

2. Zeichen:
Einschließen in Apostrophe. Z. B. 'a‘, '$‘. Nicht druckbare Zeichen werden als sog. Escape-Sequenzen angegeben.

'\n‘	neue Zeile (New Line NL)	'\‘‘	Apostroph
'\r‘	Wagenrücklauf (Carriage Return CR)	'\“‘	Anführungszeichen
'\t‘	horizontaler Tabulator	'\a‘	Alarm
'\v‘	vertikaler Tabulator	'\?‘	Fragezeichen
'\b‘	Leerschritt rückwärts (Backspace)	'\nnn‘	beliebiges Zeichen; nnn ist Oktalzahl
'\f‘	neue Seite (Form Feed FF)	'\xhh‘	beliebiges Zeichen; hh ist Hexadezimalzahl
'\\‘	Backslash (\)	'\0‘	Nullzeichen (0x00). Endekennung von Zeichenketten

3. Zeichenketten:
Einschließen in Anführungszeichen. Z. B. “Das ist eine Zeichenkette“. Zeichenketten dürfen Escape-Sequenzen enthalten. Sie werden ohne Apostrophe angegeben. Beispiel: “Zeichenkette/t mit Escape-Sequenz“.

4. Konstante Variable:
Mit Schlüsselwort CONST.

Beispiel:
```
const int temperatur = 18;
```

Ein- und Ausgabe mit E-A-Ports (Borland; DOS.H):

```
int-Variable = inport (Portadresse);          -- zwei Bytes
int-Variable = inportb (Portadresse);         -- ein Byte

outport (Portadresse, Datenwort);             -- zwei Bytes
outportb (Portadresse, Datenbyte);            -- ein Byte
```

Funktionen deklarieren:

Typ der Rückgabe-Variablen Funktionsname (Parametertyp Parametername, Parametertyp, Parametername usw.)

{

-- Deklaration der lokalen Variablen --

-- was zu tun ist ---

}

Wenn nichts zurückzugeben ist: Schlüsselwort VOID.

Wenn nichts zu übergeben ist: Funktionsname ().

Schleifen:

1. WHILE-Schleife:

while (Umlaufbedingung)

{

-- Schleifenkörper --

}

Die Umlaufbedingung wird am Anfang geprüft. Schleife wird durchlaufen, solange Umlaufbedingung erfüllt (= logisch wahr). Schleifen wird nie durchlaufen, wenn Umlaufbedingung schon zu Anfang nicht erfüllt (= logisch falsch).

2. FOR-Schleife:

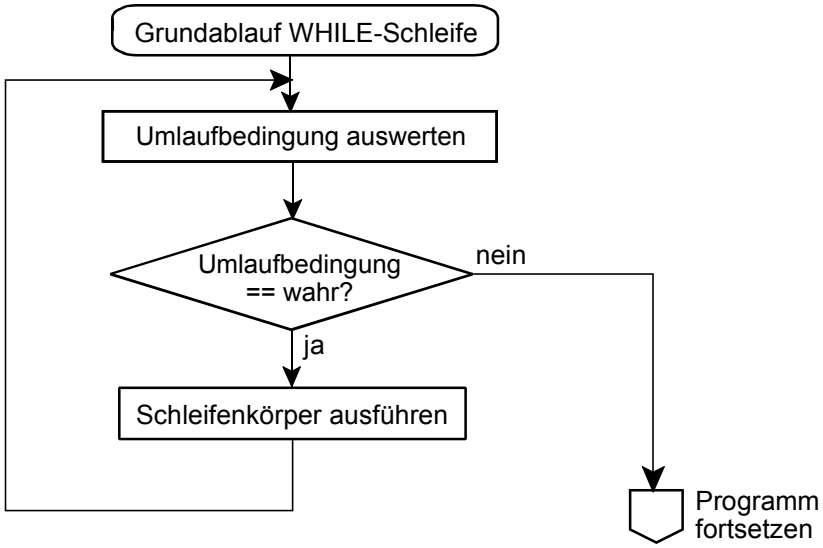
for (Laufvariable = Anfangswert, Umlaufbedingung, Wertberechnung für Laufvariable)
{

-- Schleifenkörper --

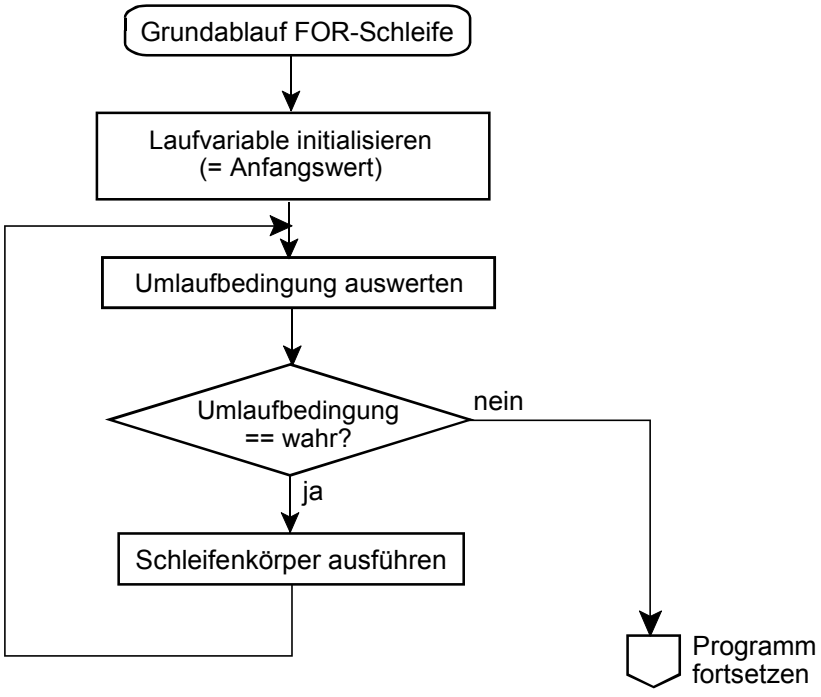
}

Zu Beginn der Schleife wird Laufvariable initialisiert. Dann wird Umlaufbedingung geprüft. Schleife wird durchlaufen, wenn Umlaufbedingung erfüllt (= logisch wahr). Nach dem Durchlauf wird der neue Wert der Laufvariablen berechnet. Schleifen wird nie durchlaufen, wenn Umlaufbedingung schon zu Anfang nicht erfüllt (= logisch falsch).

```
while (Umlaufbedingung)
{
-- Schleifenkörper --
}
```




```
for (Laufvariable = Anfangswert, Umlaufbedingung, Wertberechnung für Laufvariable)
{
-- Schleifenkörper --
}
```



3. DO-WHILE-Schleife:

```
do
{

-- Schleifenkörper --

}

while (Umlaufbedingung);
```

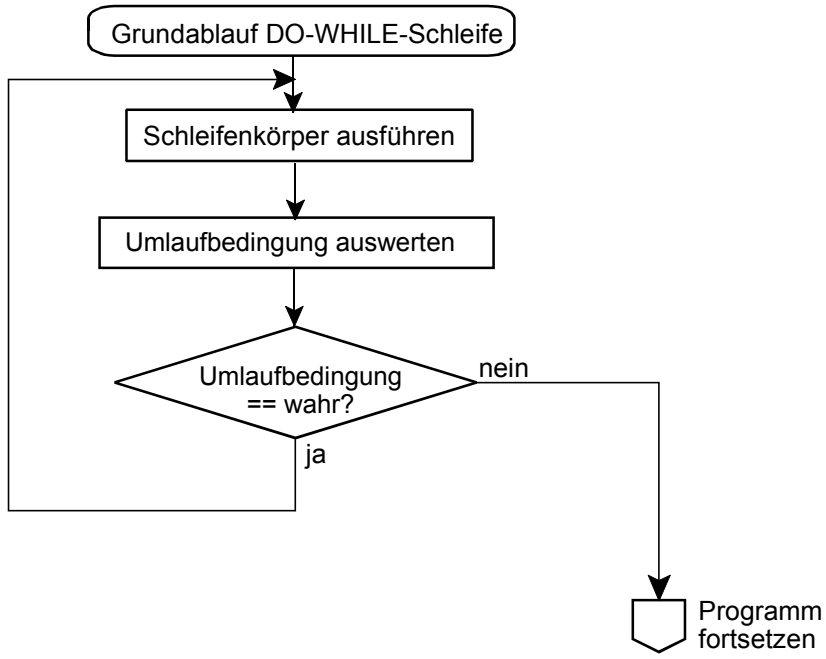
Die Umlaufbedingung wird erst am Ende geprüft. Schleife wird somit wenigstens einmal durchlaufen. Schleife wird weiterhin durchlaufen, solange Umlaufbedingung erfüllt (= logisch wahr)

```
do
{

-- Schleifenkörper --

}

while (Umlaufbedingung);
```



Vorzeitiges Verlassen von Schleifen:	break
Vorzeitig zum Test der Schleifenbedingung zurück:	continue
Aus der Schleife raus und irgendwo hin:	goto Sprungziel

Die ewige Schleife (Beispiel):

```
do
{

-- Schleifenkörper --

}

while (1);        -- Endebedingung muß immer erfüllt = logisch wahr sein.
```

Verzweigungen:

```
if (Bedingung) auszuführende Anweisung;

if (Bedingung)
{

-- auszuführende Anweisungen --

}

if (Bedingung) auszuführende Anweisung;

        else auszuführende Anweisung;

if (Bedingung)
{

-- auszuführende Anweisungen --

}
        else
{

-- auszuführende Anweisungen --

}
```

Nach else kann eine weitere if-Anweisung folgen usw. (beliebige Tiefe).

Fallunterscheidung:

```
switch (Fallvariable)
{
case 1. Testwert:

auszuführende Anweisungen;

case 2. Testwert:

auszuführende Anweisungen;

usw.

}
```

Die Fallvariable muß eine ganze Zahl sein (Wert oder Ausdruck, der den Wert berechnet).

Die Testwerte müssen Konstanten sein (echte Konstanten oder Ausdrücke, aus denen der Compiler einen konstanten Wert berechnen kann).

Verlassen der Fallunterscheidung: `break;`

Achtung: Steht nach den ausführbaren Anweisungen eines Testfalles kein `break;`, so wird der nächste Testfall untersucht!

Pauschalbehandlung aller Werte, die von den mit `case` angegebenen Testfällen nicht erfaßt werden: `default:`

Logische Werte:

False = nicht erfüllt = logisch 0: Zahlenwert = 0

True = erfüllt = logisch 1: Zahlenwert ungleich 0

Vergleichen:

==	gleich	<	kleiner
!=	ungleich	>	größer
<=	kleiner oder gleich	>=	größer oder gleich

Logische Verknüpfungen zu Testzwecken:

&&	UND
||	ODER
!	NICHT

Achtung:
Diese Operatoren befassen sich nur mit den logischen Werten gemäß obiger Zuordnung. Sie führen keine bitweisen Verknüpfungen aus. Ein Abtesten von Bitmustern ist somit nicht möglich!

Wahrheitstabelle:

A	**B**	**A && B**	**A || B**	**! A**
= 0	= 0	0	0	1
= 0	!= 0	0	1	1
!= 0	= 0	0	1	0
!= 0	!= 0	1	1	0

Beispiel:
Es soll geprüft werden, ob in der Variablen n das Bit 7 gesetzt ist.

Falsch:

```
IF (n && 0x80)        -- sofern n ungleich 0, ergibt sich stets der logische Wert 1.
```

Richtig:

```
IF ((n & 0x80) !=  0)  -- hier wird die bitweise Verknüpfung wirklich ausgeführt,
                       -- und das Verknüpfungsergebnis wird mit Null verglichen.
```

Operationen (a op **p):**

+	Addieren	a + b	%	Divisonsrest (modulo)	a % b
-	Subtrahieren	a - b	<<	Linksverschieben um b Bits	a << b
*	Multiplizieren	a * b	>>	Rechtsverschieben um b Bits	a >> b
/	Dividieren	a / b	&	UND-Verknüpfung	a & b
|	ODER-Verknüpfung	a | b	~	Negation	~ a
^	XOR-Verknüpfung	a ^ b		0	

Zuweisungen (a := a op **b):**

+=	Addieren	a += b (a = a + b)	%=	Divisonsrest (modulo)	a %= b (a = a % b)
-=	Subtrahieren	a -= b (a = a - b)	<<=	Linksverschieben	a <<= b (a = a << b)
*=	Multiplizieren	a *= b (a = a * b)	>>=	Rechtsverschieben	a >>= b (a = a >> b)
/ =	Dividieren	a /= b (a = a / b=	&=	UND-Verknüpfung	a &= b (a = a & b)
|=	ODER-Verknüpfung	a |= b a = a | b)	^=	XOR-Verknüpfung	a ^= b (a = a ^ b)

Nützliche Kleinigkeiten zum Debugging:

Ausgabe:
printf (Zeichenkette); -- die Zeichenkette kann Steuerzeichen enthalten.

printf (Zeichenkette mit n Platzhaltern, n Werte bzw. Variable, die an den Stellen der Platzhalter eingefügt werden); -- die Zeichenkette kann Steuerzeichen und Platzhalter enthalten.

Eingabe:
scanf (Zeichenkette mit n Platzhaltern, n Variable, die gemäß den Platzhaltern mit eingegebenen Werten versorgt werden);

-- normaler Text darf enthalten sein, wird aber nicht dargestellt.

Steuerzeichen:

\n	neue Zeile	\t	Tabulator
\r	zum Zeilenanfang	\b	ein Zeichen zurück (Backspace)
\a	Piepston abgeben (Alert)	\‘	einfaches Hochkomma
\\	\ = Backslash	\“	doppeltes Hochkomma
%%	% = Prozentzeichen		

Platzhalter:

%i,	Ganzzahl (Integer)	%c	Ganzzahl als ASCII-Zeichen (Zahl ist Index in die Codetabelle)
%o	Ganzzahl in oktaler Darstellung	%x	Ganzzahl in hexadezimaler Darstellung
%f	Gleitkommmazahl	%lf	doppelt genaue Gleitkommazahl
%Lf	extrem genaue Gleitkommazahl	%s	Zeichenkette
%u	vorzeichenlose Zahl	%d	Ganzzahl

Weitere Bildschirmfunktionen (Borland):

Bildschirm löschen: clrscr ();

Auf Zeile x und Spalte y positionieren: gotoxy (x,y);

Zeichen von Tastatur einlesen (ohne Darstellung): Integer-Variable = getch();

Zeichen auf Bildschirm ausgeben: Integer-Variable = putch (Zeichencode);

-- Zeichencode ist vom Typ Integer. Rückgabe: der geschriebene Zeichenccode oder der Code von EOF (im Fehlerfall).