Programmieren in C
für Elektrotechniker

Kapitel 4: Kontrollstrukturen

- **Aussagenlogik**
- Blöcke
- Selektion
- Iterationen
- Sprünge

Grundlagen

- **Universelles Programm**
  - Programmverlauf abhängig von aktuellen Werten der Daten.
  - Jede Programmiersprache besitzt hierfür Kontrollstrukturen.
    - In Abhängigkeit der Bewertung von Ausdrücken können Anweisungen übergangen oder ausgeführt werden.
    - Selbst auf Prozessorebene (Maschinencode) gibt es Befehl wie z.B.
      `beq $1, $2, Marke` (branch on equal)
      → *falls Inhalt von Register* `$1` *gleich Inhalt von Register* `$2`,
      springe zur Programmstelle „`Marke`*“ – ansonsten weiter beim nächsten Befehl*

Grundlagen

▪ Aussagenlogik, Boolesche Algebra

- Verzweigungen im Programmcode basieren auf der Auswertung von Bedingungen *(booleschen Ausdrücken)*.
- hier:
  Grundlagen der Aussagenlogik (Boolesche Algebra).
  - Ziel: Verständnis und Formulierung von Bedingungen.
  - Keine Optimierung komplexer boolescher Ausdrücke.
  - Mehr in Vorlesung „Architektur digitaler Systeme I“.

- Aussagen: Sätze, die mit „wahr“ oder „falsch“ beantwortet werden können.
  Aussageverknüpfungen: Aussagen komplexerer Natur.
- Boolesche Algebra (Boole, 1854): Formalisierung von Aussageverknüpfungen → Aussagenlogik
- Schaltalgebra (Shannon, 1938): Formalisierung von Schalterverbindungen (gleicher Formalismus)

▪ Aussagenlogik, Boolesche Algebra

- In der Aussagenlogik kennt man:
  - Konstanten: 0 und 1 (entspr. den Aussagen „falsch“ und „wahr“)
  - Variablen: $x \in IB = \{0, 1\}$
    entspr. oft logischen Aussagen, z.B.
    x = 1 → „Die Sonne scheint.“
    x = 0 → „Die Sonne scheint nicht.“
  - Funktionen: *(Ausdrücken)*
    logische Verknüpfungen von Variablen
    $f: IB^n \in IB$, z.B. $y = f(x_1, x_2)$

- Grundfunktionen

a) Negation, NOT (NICHT): $y = f(x) = \overline{x}$ (bzw. $\neg x$)

x	$\overline{x}$
0	1
1	0

Es gilt: $\overline{\overline{x}} = x$

- Grundfunktionen

b) Disjunktion, OR (ODER), Addition: $y = f(x_1, x_2) = x_1 + x_2$ (bzw. $x_1 \vee x_2$)

x_1	x_2	$x_1 + x_2$
0	0	0
0	1	1
1	0	1
1	1	1

Es gilt: $x + 1 = 1$ $x + x = x$
$x + 0 = x$ $x + \overline{x} = 1$

c) Konjunktion, AND (UND), Multiplikation: $y = f(x_1, x_2) = x_1 \cdot x_2$ (bzw. $x_1 \wedge x_2$)

x_1	x_2	$x_1 \cdot x_2$
0	0	0
0	1	0
1	0	0
1	1	1

Es gilt: $x \cdot 0 = 0$ $x \cdot x = x$
$x \cdot 1 = x$ $x \cdot \overline{x} = 0$

Boolesche Ausdrücke enthalten nur die Verknüpfungen NICHT, UND, ODER

allgemeine logische Ausdrücke: auch weitere Verknüpfungen, u.a.:
- Implikation ($\rightarrow$)
- Äquivalenz ($\leftrightarrow$, $\equiv$)
- Antivalenz ($\oplus$)

$\rightarrow$ lassen sich auf NICHT, UND, ODER abbilden *(s.u.)*

Auswertungsreihenfolge:
1. Negation
2. Konjunktion
3. Disjunktion
(4. Implikation
5. Äquivalenz
6. Antivalenz)

Alle null- und einstelligen Verknüpfungen $f: IB \rightarrow IB$

a) „Verknüpfungen“ ohne Aussagevariablen

	y	Bezeichnung
1.	0	Kontradiktion (nie)
2.	1	Tautologie (immer)

Alle null- und einstelligen Verknüpfungen

f: IB → IB

b) „Verknüpfungen“ mit einer Aussagevariablen y = f(x)

x	Kontradiktion y	Identität y	Negation y	Tautologie y
0	0	0	1	1
1	0	1	0	1

bzw. *(Tabelle um 90^0 gedreht)*

Funktionswert für	x =	0	1	Funktion y = f(x)	Bezeichnung
	y =	0	0	y = 0	Nullfunktion / Kontradiktion
		0	1	y = x	Identität
		1	0	$y = \overline{x}$	Negation
		1	1	Y = 1	Einsfunktion / Tautologie

Programmieren in C

Alle zweistelligen Verknüpfungen

f_1: IB x IB → IB, ..., f_{16}: IB x IB → IB

$$y = f(x_1, x_2)$$

x_1	x_2	y															
0	**0**	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	**1**	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	**0**	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	**1**	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

Alle zweistelligen Verknüpfungen

f_1: IB x IB→ IB, ..., f_{16}: IB x IB→ IB

Funktionswert für	x_1 =	0	1	0	1	Funktion $y = f(x_1, x_2)$	Bezeichnung
	x_2 =	0	0	1	1		
	y =	0	0	0	0	0	Nullfunktion
		0	0	0	1	$x_1 \cdot x_2$	Konjunktion, AND
		0	0	1	0	$x_2 \not\rightarrow x_1 = \overline{x}_1 x_2$	Inhibit
		0	0	1	1	x_2	Identität
		0	1	0	0	$x_1 \not\rightarrow x_2 = x_1 \overline{x}_2$	Inhibit
		0	1	0	1	x_1	Identität
		0	1	1	0	$x_1 \oplus x_2 = x_1 \overline{x}_2 + \overline{x}_1 x_2$	Antivalenz, XOR, EXOR
		0	1	1	1	$x_1 + x_2$	Disjunktion, OR
		1	0	0	0	$\overline{x_1 + x_2}$	NOR (Peirce-Funktion, weder-noch)
		1	0	0	1	$x_1 \equiv x_2 = x_1 x_2 + \overline{x}_1 \overline{x}_2$	Äquivalenz
		1	0	1	0	$\overline{x}_1$	Negation, NOT
		1	0	1	1	$x_1 \rightarrow x_2 = \overline{x}_1 + x_2$	Implikation: wenn x_1, dann x_2 (ist falsch für x_1=1, x_2=0)
		1	1	0	0	x_2	Negation
		1	1	0	1	$x_2 \rightarrow x_1 = x_1 + \overline{x}_2$	Implikation
		1	1	1	0	$\overline{x_1 \cdot x_2}$	NAND (Sheffer-Funktion)
		1	1	1	1	1	Einsfunktion

Rechenregeln

a) Axiome

- Kommutativgesetze: $x_1 + x_2 = x_2 + x_1$
  $x_1 \cdot x_2 = x_2 \cdot x_1$

- Distributivgesetze: $x_1 \cdot (x_2 + x_3) = x_1 x_2 + x_1 x_3$
  $x_1 + (x_2 \cdot x_3) = (x_1 + x_2) \cdot (x_1 + x_3)$

- Neutrale Elemente: $x + 0 = x$ → *s.o.*
  $x \cdot 1 = x$

- Komplement: $x + \overline{x} = 1$ → *s.o.*
  $x \cdot \overline{x} = 0$

b) weitere Rechenregeln

- Maximale Elemente: $x + 1 = 1$ → *s.o.*
  $x \cdot 0 = 0$
- Idempotenzgesetze: $x + x = x$ → *s.o.*
  $x \cdot x = x$
- Assoziativgesetze: $x_1 + (x_2 + x_3) = (x_1 + x_2) + x_3$
  $x_1 \cdot (x_2 \cdot x_3) = (x_1 \cdot x_2) \cdot x_3$
- Verschmelzungsgesetze (Absorptionsgesetze):
  $x_1 \cdot (x_1 + x_2) = x_1$
  $x_1 + (x_1 \cdot x_2) = x_1$
- De Morgansche Gesetze: $\overline{x_1 + x_2} = \overline{x_1} \cdot \overline{x_2}$
  $\overline{x_1 \cdot x_2} = \overline{x_1} + \overline{x_2}$

Programmieren in C

Boolesche Algebra

Definition

Sei IB eine Menge mit zwei verschiedenen Elementen 1, 0 ∈ IB,
über der eine unäre Operation −
und zwei binäre Operationen + und · definiert sind.

Falls die folgenden Gesetze *(Axiome)* gelten,
nennt man (IB, +, ·, −, 1,0) eine Boolesche Algebra:

- Kommutativität
- Distributivität
- neutrale Elemente
- Komplement

Beispiel für Vereinfachung eines Ausdrucks (algebraische Minimierung)

$x_1 + \overline{x}_1 x_2 + \overline{x}_1 \overline{x}_2 x_3$

=	$(x_1 + \overline{x}_1) \cdot (x_1 + x_2) + \overline{x}_1 \overline{x}_2 x_3$	(Distributivgesetz: a+bc = (a+b)(a+c))
=	$1 \cdot (x_1 + x_2) + \overline{x}_1 \overline{x}_2 x_3$	(Komplement: $a+\overline{a} = 1$)
=	$(x_1 + x_2) + \overline{x}_1 \overline{x}_2 x_3$	(neutrales Element: $1 \cdot a = a$)
=	$(x_1 + x_2) + \overline{(x_1 + x_2)} \cdot x_3$	(de Morgan: $\overline{a+b} = \overline{a} \cdot \overline{b}$)
=	$((x_1 + x_2) + \overline{(x_1 + x_2)}) \cdot ((x_1 + x_2) + x_3)$	(Distributivgesetz: s.o.)
=	$1 \cdot (x_1 + x_2 + x_3)$	(Komplement: s.o.)
=	$x_1 + x_2 + x_3$	(neutrales Element: s.o.)

Programmieren in C

für Elektrotechniker

Kapitel 4: Kontrollstrukturen

- Aussagenlogik
- **Blöcke**
- Selektion
- Iterationen
- Sprünge

Blöcke (Vertiefung)

Kontrollstruktur der Sequenz

- Ein Block kann als zusammengesetzte Anweisung betrachtet werden und ist überall dort erlaubt, wo eine Anweisung stehen darf.

```
{
   Vereinbarungen
   Anweisung_1
   Anweisung_2
   . . .
   Anweisung_n
}
```

- Vereinbarungen: Deklarationen und Definitionen von Variablen.
  - außerhalb von Funktionen (→ global)
  - vor erster Anweisung einer Funktion oder eines Blocks
- Anweisungen in Blöcken werden sequentiell ausgeführt.
- Ein Block zählt syntaktisch als eine einzige Anweisung.
- Kein Semikolon nach schließender Klammer ‚}'

Verwendung

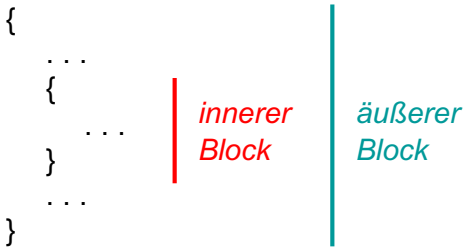
- Rumpf einer Funktion.
- Sequenz von Anweisungen überall dort, wo eine Anweisung erlaubt ist
  (z.B. if- und else-Zweig einer if-Anweisung, s.u.).
- Logische Gliederung von Anweisungsfolgen
  (ggf. mit eingeschränkter Sichtbarkeit von Variablen, s.u.).

Deklarationen am Anfang eines Blocks

- Definition eines Datentyps *(Aufzählungstyp, Struktur, Union).*
- Vorwärtsdeklaration eines Datentyps
  (falls Datentypen zyklisch aufeinander verweisen, z.B. bei Pointer – später mehr).
- Definition eines neuen Datentyps mit „typedef"
  (Alias-Namen für Datentypen, um Schreibarbeit zu sparen – später mehr).
- Deklarationen von Variablen
  (z.B. bei externen Variablen – später mehr).
- Deklaration von Funktionen
  (Funktionsprototypen für Funktionen, die nach ihrer Verwendung definiert werden – später mehr).

Schachtelung

- Blöcke können geschachtelt werden
  (Blöcke dort erlaubt, wo Anweisungen erlaubt sind).



- In C können in jedem (inneren) Block Vereinbarungen durchgeführt werden.
  → Vereinbarungen innerhalb eines Blocks sind lokal für diesen Block.
- Lokal definierte Variablen und lokal deklarierte Namen sind nur innerhalb des Blocks sichtbar und in einem umfassenden, äußeren Block unsichtbar.
- In einem äußeren Block definierte Variablen und deklarierte Namen sind in inneren Blöcken sichtbar.

Schachtelung

- Beispiel

→ Programmausgabe bei Eingabe 1 bzw. -1?

```
Gib a ein: 1
y hat den Wert __
x hat den Wert __
Der Wert von a ist __

Gib a ein: -1
y hat den Wert __
x hat den Wert __
Der Wert von a ist __
```

```
int y = 2;

int main (void)
{
    int x = 5;
    int a;

    printf ("Gib a ein: ");
    scanf ("%d", &a);

    if (a > 0)
    {
        int y = 4;
        printf ("\ny hat den Wert %d", y);
        printf ("\nx hat den Wert %d", x);
    }
    else
    {
        int x = 4;
        printf ("\ny hat den Wert %d", y);
        printf ("\nx hat den Wert %d", x);
    }
    printf ("\nDer Wert von a ist %d", a);
    return 0;
}
```

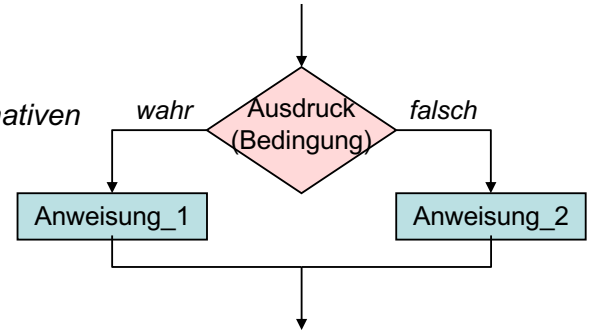
Programmieren in C
für Elektrotechniker

Kapitel 4: Kontrollstrukturen

- Aussagenlogik
- Blöcke
- **Selektion**
- Iterationen
- Sprünge

Selektion

→ Auswahl zwischen zwei oder mehreren Alternativen



- **Einfache Auswahl: if-else**

```
if (Ausdruck)
   Anweisung_1
else
   Anweisung_2
```

- Zunächst wird der Ausdruck in der Klammer berechnet (→ Bedingung)
  - Ergebnis ≠ 0 (≅ wahr): Ausführung von **`Anweisung_1`**.
  - Ergebnis = 0 (≅ falsch): Ausführung von **`Anweisung_2`**.

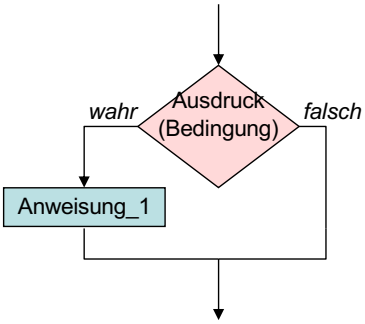
```
int a;
if (a != 0) ...
```

entspr.

```
int a;
if (a) ...
```

besser: boolean

▪ Einfache Auswahl: if-else



- else-Zweig ist optional

```
if (a)
   Anweisung
```

→ bedingte Anweisung

- Geschachtelte if-else-Anweisungen
  - Anweisungen in **`if`**- und **`else`**-Zweigen können wieder **`if-else`**-Auswahlen sein.
  - Beispiel:

```
max (a, b, c);
if (a > c)
   if (b > a) max = b;
   else max = a;
else
   if (b > c) max = b;
   else max = c;
```


▪ Einfache Auswahl: if-else

- Zugehörigkeit von **`else`**-Zweigen
  - Beispiel:

```
if (a)
   if (b > c)
      printf ("Ausgabe 1");
else
   printf ("Ausgabe 2");
```

Ausgabe bei:
`a = 0`
`b = 1`
`c = 2 ?`

  - Mehrdeutigkeit aufgrund optionaler **`else`**-Zweige.
  - Ein **`else`**-Zweig gehört immer zum letzten freien **`if`**
    (für das es noch kein `else` gibt).
  - Beispiel oben: - **`else`** gehört zum inneren **`if`**
    - Programmstrukturierung hierfür uninteressant
  - Lösung: Strukturierung durch Block-Klammern

```
if (a)
{
   if (b > c)
      printf ("Ausgabe 1");
}
else
   printf ("Ausgabe 2");
```

Ausgabe bei:
`a = 0`
`b = 1`
`c = 2 ?`

Selektion

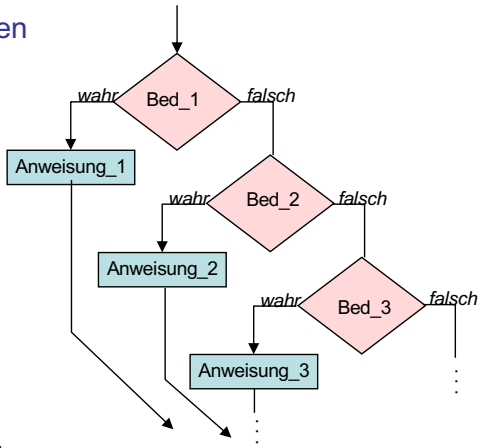
→ Auswahl zwischen zwei oder mehreren Alternativen

▪ Mehrfach-Auswahl

- Möglichkeit 1: Mehrfache **`if-else`**-Auswahlen
  - Beispiel:

```
if (Bedingung_1)
    Anweisung_1
else if (Bedingung_2)
    Anweisung_2
else if (Bedingung_3)
    Anweisung_3
...
else if (Ausdruck_n)
    Anweisung_n
else /* der else-Zweig */
    Anweisung_else
```

Einrücken eigentlich nicht korrekt!



  - Letzter **`else`**-Zweig ist optional
    - behandelt i.d.R. alle weiteren Fälle, die nicht explizit behandelt werden
    - häufig Fehlerfall
    - kann entfallen.

▪ Mehrfach-Auswahl

- Möglichkeit 2: **`switch`**-Alternativen
  - Mehrfachauswahl über Integer-Ausdrücke.
  - Beispiel:

```
switch (Ausdruck)
{
    case k1: Anweisungen_1
             break; /* optional */
    case k2: Anweisungen_2
             break;
    ...
    case kn: Anweisungen_n
             break;
    default: /* optional */
             Anweisungen_default
}
```

  - Nach Auswertung von **`Ausdruck`** Sprung zur passenden Marke *(alle Marken müssen unterschiedlich sein).*
  - Stimmt keine Marke überein: Sprung zur **`default`**-Marke (bzw. hinter die **`switch`**-Anweisung).
  - Reihenfolge der Marken und **`default`** ist beliebig.

Mehrfach-Auswahl

- Möglichkeit 2: **`switch`**-Alternativen
  - Mehrfachauswahl über Integer-Ausdrücke.
  - Beispiel:

```
switch (Ausdruck)
{
    case k1:  Anweisungen_1
              break; /* optional */
    ...
    default: /* optional */
              Anweisungen_default
}
```

  - **`case`**-Marken **`k1, .., kn`**:
    ganzzahlige Konstanten oder ganzzahlige konstante Ausdrücke.
  - Auch mehrere Marken vor einer Anweisung möglich:
    `case 1: case 5: case 7: Anweisungen; break;`
  - Nach Sprung zu einer Marke werden alle Anweisungen bis **`break`** oder Ende der **`switch`**-Anweisung ausgeführt (auch Anweisungen hinter anderen Marken).
    `break`: Spring hinter **`switch`**-Anweisung.

Programmieren in C Auswahl

Mehrfach-Auswahl

- Möglichkeit 1: Mehrfache if-else-Auswahlen
- Möglichkeit 2: **`switch`**-Alternativen
  - Unterschiede der beiden Ansätze:
    - **`switch`**: nur Vergleich von Integer-Werten
      (if-else: Auswertung beliebiger Bedingungen/Ausdrücke).
    - **`switch`**: i.d.R. schneller (→ Sprungtabellen möglich).
    - **`switch`**: i.d.R. übersichtlicher.

Mehrfach-Auswahl

- Beispiele

```
int main (void)
{
   enum color {RED, GREEN, BLUE};
   enum color col = GREEN;
   switch (col)
   {
      case RED:
         printf(" RED");
      case GREEN:
         printf(" GREEN");
      case BLUE:
         printf(" BLUE");
      default:
         printf(" undefined color!");
   }
   return 0;
}
```

Ausgabe?

Mehrfach-Auswahl

- Beispiele

```
int main (void)
{
   enum color {RED, GREEN, BLUE};
   enum color col = GREEN;
   switch (col)
   {
      case RED:
         printf(" RED");
         break;
      case GREEN:
         printf(" GREEN");
         break;
      case BLUE:
         printf(" BLUE");
         break;
      default:
         printf(" undefined color!");
   }
   return 0;
}
```

Ausgabe?

Mehrfach-Auswahl

- Beispiele

```
int main (void)
{
   int zahl;
   printf ("\nEingabe: ");
   scanf ("%d",&zahl);
   switch (zahl)
   {
   case 2: case 4:
      printf ("\nGerade Zahl zw. 1 und 5\n");
      break;
   case 1: case 3: case 5:
      printf ("\nUngerade Zahl zw. 1 und 5\n");
      break;
   default:
      printf ("\nKeine Zahl zw. 1 und 5\n");
   }
   return 0;
}
```

Mehrere Marken für eine Anweisungsfolge.

Mehrfach-Auswahl

- Beispiel

```
...
switch (zahl)
{
   case 1: case 3: case 5: case 7: case 11: case 13:
      printf ("Primzahl und");
   case 9: case 15:
      printf („ungerade Zahl");
      break;
   default:
      printf („keine ungerade Zahl zw. 1 und 15");
}
```

Ausgabe bei zahl = 4, 9 und 5?

Mehrfach-Auswahl

- Tipp: Jeder Fall einer **`switch`**-Anweisung sollte mit **`break`** abgeschlossen werden.
- Beispiel

```
...
switch (zahl)
{
   case 1: case 3: case 5: case 7: case 11: case 13:
      printf ("Primzahl und ungerade Zahl");
      break;
   case 2:
      printf ("Primzahl und gerade Zahl");
      break;
   case 9: case 15:
      printf („ungerade Zahl");
      break;
   default:
      printf („keine ungerade Zahl zw. 1 und 15");
}
```


Programmieren in C

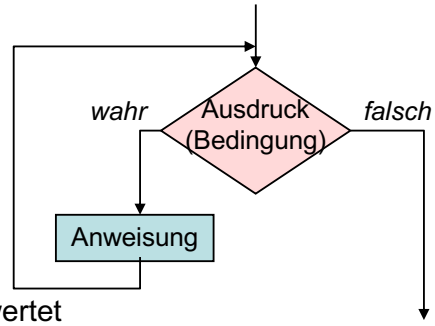
für Elektrotechniker

Kapitel 4: Kontrollstrukturen

- Aussagenlogik
- Blöcke
- Selektion
- **Iterationen**
- Sprünge

Iteration (Schleifen)

→ wiederholte Ausführung von Anweisungen



▪ Abweisende while-Schleife

```
while (Ausdruck)
    Anweisung
```

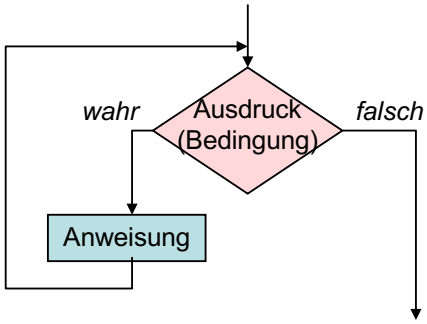
- **`Ausdruck`** wird vor Schleifendurchlauf ausgewertet
  → abweisende Schleife
- Problem: Endlosschleife möglich, falls **`Ausdruck`** nie zu falsch ausgewertet wird.
- Beispiel:

```
int main (void)
{
   int lv = 0;
   while (lv < 10)
   {
      printf ("%d", lv);
      if (lv < 9)
         printf (", ");
      lv = lv + 1;
   }
   return 0;
}
```

Ausgabe?

Iteration (Schleifen)

→ wiederholte Ausführung von Anweisungen



▪ Abweisende for-Schleife

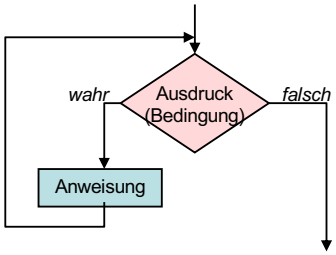
```
for (A1; A2; A3)
   Anweisung
```

- Entspricht

```
A1;
while (A2)
{
    Anweisung;
    A3;
}
```

- 3 Schritte:
  - **`A1`**: Initialisierung der Laufvariablen.
  - **`A2`**: Prüfung der Schleifenbedingung.
  - Ggf. **`Anweisung`** (→ Schleifenrumpf).
  - **`A3`**: Ggf. Anweisung zur Änderung des Werts der Laufvariablen

  (falls kein Abbruch durch **`A2`**)



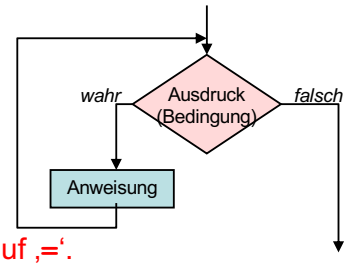
▪ Abweisende for-Schleife

```
for (A1; A2; A3)
   Anweisung
```

- Typisch: - **A1**, **A3**: Zuweisung an eine Laufvariable
  (aber alle Anweisungen/Nebeneffekte erlaubt).
  - **A2**: Bedingung über Wert der Laufvariablen.
- Nach Verlassen der Schleife hat die Laufvariable den zuletzt berechneten Wert.
- Beispiele:

```
int main (void)
{
    int i;
    for (i=0; i<10; i++)
    {
        printf ("%d", i);
        if (i<9) printf (", ");
    }
    return 0;
}
```

```
int main (void)
{
    int i;
    for (i=2; i>-1; i--)
    {
        printf ("%d", i);
        if (i>0) printf (", ");
    }
    return 0;
}
```

▪ Abweisende for-Schleife

```
for (A1; A2; A3)
   Anweisung
```

- Tipp: **A2**: Auf ‚<' und ‚>' prüfen ist meist sicherer als auf ‚=‘.

```
for (float lv = 1.0; lv != 1.7; lv = lv + 0.1)
```

→ je nach Rundung wird **lv** nie genau **1.7**.

- besser: `for (float lv=1.0; lv < 1.7; lv=lv+0.1)`
- noch besser: Keine **float**-Laufvariable. ☺

- **for**-Schleife häufig zur Iteration über Arrays

```
int main (void)
{
    int index;
    int array1 [20];
    int array2 [20];
    for (index = 0; index < 20; index++)
    {
        array1[index] = index;
        array2[index] = array1[index] * 2;
    }
    return 0;
}
```

▪ Abweisende for-Schleife

- Weitere Beispiele

```
int main (void)
{
    int i;
    for (i = 12; i >= 0; i = i - 2)
    {
        printf ("%d",i);
        if (i > 0) printf (", ");
    }
    printf ("\n");
    for (i = -1; i >= -13; i = i - 2)
    {
        printf ("%d", i);
        if (i > -12) printf (", ");
    }
    printf ("\n");
    for (i = 0; i <= 6; i = i + 1)
    {
        printf ("%d", i * i);
        if (i < 6) printf (", ");
    }
    printf ("\n");
    return 0;
}
```

Ausgabe?

Programmieren in C Auswahl

▪ Abweisende for-Schleife

- Weitere Beispiele

```
int main (void)
{
    int i1, i2;
    int teiler;
    printf ("_____________:\n");
    for (i1 = 2, teiler = 0;
         i1 <= 101;
         i1 = i1 + 1, teiler = 0)
    {
        for (i2 = 2; i2 < i1; i2 = i2 + 1)
        {
            if (i1 % i2 == 0) teiler = 1;
        }
        if (teiler == 0)
        {
            printf ("%d", i1);
            if (i1 < 101) printf (", ");
        }
    }
    return 0;
}
```

Ausgabe?

▪ Abweisende for-Schleife

- Weitere Beispiele

→ ***optimiert***

```
int main (void)
{
    int i1, i2;
    int teiler;
    printf ("_____________:\n");
    for (i1 = 2, teiler = 0;
         i1 <= 101;
         i1 = i1 + 1, teiler = 0)
    {
        for (i2 = 2;
             i2 * i2 <= i1 && teiler == 0;
             i2 = i2 + 1)
        {
            if (i1 % i2 == 0) teiler = 1;
        }
        if (teiler == 0)
        {
            printf ("%d", i1);
            if (i1 < 101) printf (", ");
        }
    }
    return 0;
}
```

oder: ***break***

▪ Abweisende for-Schleife

- Endlosschleife
  - Jeder der drei Ausdrücke A1, A2, A3 im Schleifenkopf kann entfallen
    → leere Anweisungen

```
for ( ; ; )
{
    . . .
}
```

keine Berechnung nach Schleifendurchlauf
leerer Ausdruck: Bedingung gilt immer als wahr
keine Initialisierung

  - Endlosschleife ohne Abbruch ist selten sinnvoll
    → break-Anweisung zum Abbruch.

auch

```
while (1)
{
    . . .
}
```

▪ Abweisende for-Schleife

- Leere Anweisungen (leerer Schleifenrumpf)

```
long (i);
for (i = 0; i < 100000; i = i + 1)
{
}
```

bzw.

```
for (i = 0; i < 100000; i = i + 1);
```

- Tipp: Da Schleifenrümpfe häufig später erweitert werden, sollte man gleich Blöcke einsetzen
  (auch für zunächst leere Anweisungen).

```
for (i = 0; i < 100000; i = i + 1);
  printf ("Bin in Warteschleife\n");
printf ("Der Wert des Zaehlers ist %d \n", i);
```

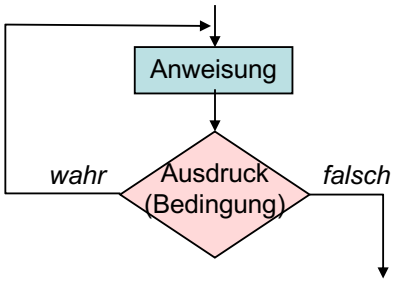
Ausgabe?

Iteration (Schleifen)

→ wiederholte Ausführung von Anweisungen

▪ Annehmende do-while-Schleife

```
do
{
   Anweisung
} while (Ausdruck);
```



- Schleifenrumpf wird vor Bedingungsprüfung (**`Ausdruck`**) ausgeführt
  → annehmende Schleife.
- Schleifenrumpf wird solange ausgeführt, wie **`Ausdruck`** wahr ist
  → solange **`Ausdruck`** nicht **`Null`**.
- Schleife wird mindestens einmal ausgeführt.

```
/*....*/
int zahl;
long summe = 0;
do
{
    scanf ("%d", &zahl);
    summe = summe + zahl;
} while (zahl);
/*....*/
```

***`Do-while`**-Schleife sinnvoll, da **`zahl`** vor Prüfung eingelesen werden muss.*

Programmieren in C
für Elektrotechniker

Kapitel 4: Kontrollstrukturen

- Aussagenlogik
- Blöcke
- Selektion
- Iterationen
- **Sprünge**

Sprunganweisungen
→ in der Regel Ausnahmefälle

break

- Beendet **`switch`**-Anweisung und (vorzeitig) **`while`**-, **`for`**-, **`do-while`**-Schleifen.
- Es wird nur die aktuelle, innerste Schleife bzw. **`switch`**-Anweisung beendet.

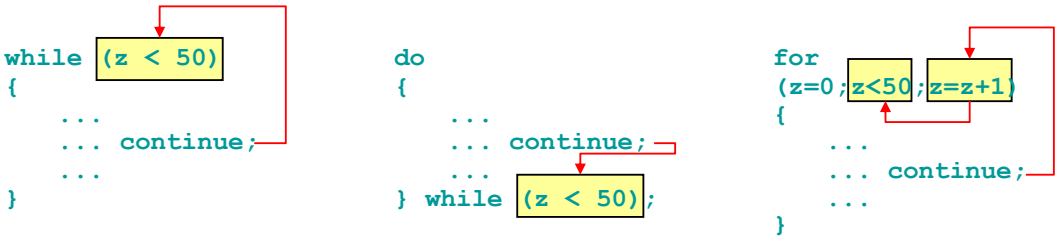
```
...
char eingabe;
for ( ; ; )
{
    eingabe = getchar(); /* ein Zeichen von Tastatur */
    if (eingabe == 'Q')
    {
        break;
    }
     ...
}
...
```

Sprunganweisungen

→ *in der Regel Ausnahmefälle*

▪ continue

- Überspringt Rest einer Schleife (Schleifenrumpf), aber verlässt die Schleife nicht.
- **`while`**- und **`do-while`**-Schleifen: **`continue`** springt direkt zur Bedingung.
- **`for`**-Schleife: **`continue`** springt zum dritten Ausdruck im Schleifenkopf (bevor die Bedingung/**`A2`** ausgewertet wird).



Sprunganweisungen

→ *in der Regel Ausnahmefälle*

▪ continue

- Beispiel

```
#include <stdio.h>

int main (void)
{
    int i = 0;
    int zahl;
    do
    {
        printf ("Geben Sie eine positive Integer-Zahl ein: ");
        i = i + 1;
        scanf ("%d", &zahl);
        if (zahl <= 0)
            continue;
        printf ("Die Zahl war: %d\n", zahl);
    } while (i < 5);
    return 0;
}
```

Sprunganweisungen

→ in der Regel Ausnahmefälle

▪ goto

- Sprung zu einer angegebenen Marke.

```
while (....)
{
    while (....)
    {
        ....
        if (Abbruch) goto Weiter_nach_Schleife;
    }
....
}
Weiter_nach_Schleife: Anweisung
....
```

- Sollte eigentlich nicht verwendet werden *(→ typisch für Assembler-Programme)*
  Dijkstra 1968: „Goto Statement Considered Harmful“
  → goto für regulären Kontrollfluss überflüssig.
- Ggf. sinnvoll im Fehlerfall zum Abbruch aus mehrfach geschachtelten Schleifen *(statt umständlich mehrere breaks mit entspr. Abfragen).*
- In C++, Java: exceptions *(→ gotos nicht mehr notwendig)*.

Programmieren in C Sprung

Sprunganweisungen

→ in der Regel Ausnahmefälle

▪ return

- Verlässt einen Funktionsaufruf (mit Rückgabewert).
- Springt hinter den Aufruf im Code des Caller.

Sprunganweisungen

→ in der Regel Ausnahmefälle

▪ Strukturierte Programmierung

- Beschränkung auf Kontrollstrukturen mit einem Eingang und einem Ausgang.
- GOTO widerspricht der Strukturierten Programmierung

Zweiter Ein- bzw. Ausgang ist bei der strukturierten Programmierung nicht erlaubt.