Kontrollstrukturen
Blöcke / Schleifen / Bedingungen

Einfache Anweisungen und Blöcke

- einfache Anweisung
  - abgeschlossen mit Semikolon ;
  - typische Fälle:
    - Deklaration, Zuweisung, Funktionsaufruf
  - Sonderfall leere Anweisung: nur ein Semikolon ;

```
int a, b;
a = b;
Summe = b + c;
Zaehler++;
Stack.Push(27);
;
```

- Block (compound statement)
  - mehrere Anw. in geschweiften Klammern { }
    - Semikolon nicht erforderlich
  - typische Fälle:
    - in `if`, `while`, `for`-Anweisung
    - Definition einer Funktion
  - Deklarationen gelten bis zum Ende des umschließenden Blocks

```
{
  int a, b;
  a = b;
  Summe = b +
c;
}
```

Überblick Kontrollstrukturen

Paradigma der
Strukturierten Programmierung:
jeder Ablauf läßt sich allein mit
Sequenz
Verzweigung
Schleife
realisieren

- strukturiert **J**
  - if ... else ... Alternative
  - switch ... case ... Fallunterscheidung
  - while ... abweisende Schleife
  - do ... while ... nicht-abweisende Schleife
  - for ... zählende Schleife
- maschinennah **L** nicht verwenden !
  - break vorzeitiger Schleifenausstieg (nur nach case ok)
  - continue überspringt Schleifenrest
  - goto beliebiger Sprung

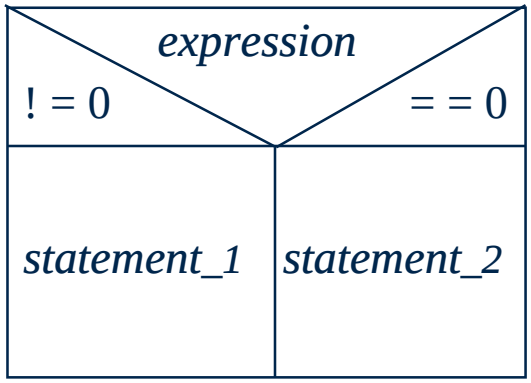
if ... else ... Alternative

- Bedingung eingeklammert
- `else` Teil ist optional
- Einrückung dient der Lesbarkeit
  - irrelevant für Compiler
  - wichtig für den Menschen
- `;` ist Teil von *statement*
  - gehört nicht zu if

if (*expression*)
  statement

---

if (*expression*)
  statement_1
else
  statement_2



Struktogramm

if ... else ... Typ der Bedingung

- Typ bool existiert nicht in C, erst in C++ (ab Visual C++ 5.0)
  Vergleichsoperationen liefern Typ int `true ≡ 1`
  `false ≡ 0`

  a) `if (`*expression*`)` ist gleichbedeutend mit
  b) `if (`*expression* `!= 0)`

- Empfehlung:
  - Form a) wenn *expression* vom Typ `bool` ist
  - Form b) wenn *expression* eine Zahl ist

if ... else ... Schachtelung

- `else` gehört immer zum **unmittelbar vorangehenden** `if`
- Vorsicht:
  der Mensch gewichtet Einrückung stärker als Klammern

è lieber ein Paar Klammern zu viel ...

richtig:

```
if (n > 0 )
   if (a > b)
      z = a;
   else
      z = b;
```

falsch: **M**

```
if (n > 0 )
   if (a > b)
      z = a;
else
   z = b;
```

hier ; erforderlich

richtig:

```
if (n > 0 ) {
   if (a > b)
      z = a;
}
else
   z = b;
```

kein ; hinter }

else if — Mehrfache Verzweigung

- Übersichtliche Form für **beliebige** Bedingungen
- begrenzt Schachtelungstiefe
- immer mit `else` abschließen (evtl. Fehlermeldung)

Syntax:

```
if ( expression_1 )
    statement_1
else if ( expression_2 )
    statement_2
else if ( expression_3 )
    statement_3
else
    statement_4
```

Beispiel:

```
if (Punkte <= 30)
    Note = 5;
else if (Punkte <= 50)
    Note = 4;
else if (Punkte <= 65)
    Note = 3;
else if (Punkte <= 80)
    Note = 2;
else
    Note = 1;
```

Bedingungsoperator ? :

- bedingte Anweisung

```
if (cond_expr)
  statement1
else
  statement2
```

- bedingter Ausdruck

```
cond_expr ? then_expr : else_expr
```

  - Auswertereihenfolge garantiert:
    erst `cond_expr`, dann entweder `then_expr` oder `else_expr`

- Beispiel zum Vergleich:

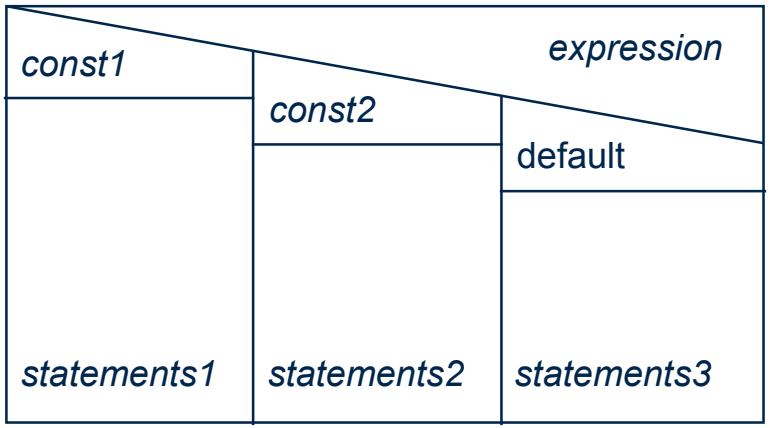
```
if (Schaltjahr)
  TageImFebruar = 29;
else
  TageImFebruar = 28;

TageImFebruar = Schaltjahr ? 29 : 28;
```

switch ... case ... Fallunterscheidung

- Übersichtliche Form abhängig von **einem** Ganzzahlwert
- mehrere `case` Marken für dieselbe Anweisung möglich
- `break` im Normalfall letzte Anweisung in *statements*
- `break` kann weggelassen werden
- `default` sollte angegeben sein (ggfs. Fehlermeldung)
- Blockklammern `{}` nicht vergessen !

```
switch ( expression ) {
    case const1: statements1
    case const2: statements2
    default: statements3
}
```

Folge von *statements* ohne Klammern möglich



switch ... case ... Beispiel

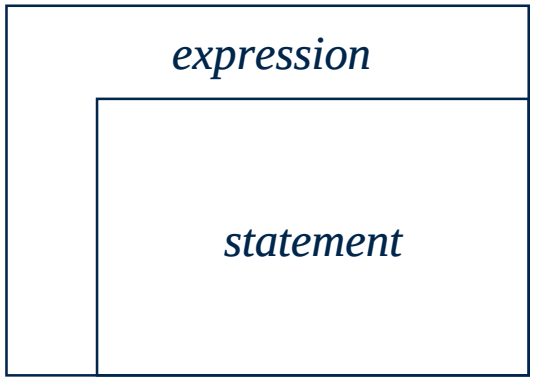
```
switch (EingabeZahl) {          // Typ int; Block einklammern !
   case 0: case 1: case 7:
      break;
   case 2: case 4: case 8:      // auch möglich: 3 case Marken
      Teiler_2++;
      break;
   case 6:
      Teiler_2++;               // fehlt hier das break ???
   case 3: case 9:              // nein: 6 ist durch 2 und 3 teilbar
      Teiler_3++;               // L zwei Zeilen Code gespart
      break;
   case 5:                      // Standard: eine case Marke
      Teiler_5++;               // irgendeine Aktion
      break;                    // J überspringt folgende cases

   default:                     // für übrige Werte von EingabeZahl
      throw "Fehler";           // J Programmierer kontrolliert
      break;                    //                sich selbst
}                               // Klammer nicht vergessen !
```

while ... abweisende Schleife

- Bedingung eingeklammert
- `;` ist Teil von *statement*
- Typ der Bedingung wie bei `if`
- *statement* wird ausgeführt solange *expression* `!=0`
- **0** .. n Schleifendurchläufe
- zur Abarbeitung verketteter Listen empfohlen

Laufbedingung

```
while ( expression )
  statement
```



Struktogramm

Grundsätzliches zu Schleifen

- i.a. Laufvariable erforderlich - identifiziert Schleifendurchlauf
  - Zaehler, Zeiger auf Liste, ...
  - Ausnahme z.B.: Schleife, die infolge einer Eingabe abbricht
- Bedingungsausdruck muß Laufvariable auswerten
  - ersten u. letzten Durchlauf beachten; (±1 Fehler) vermeiden
  - die Bedingung ist eine Laufbedingung, keine Abbruchbedingung !
- Laufvariable im Schleifenkörper Statement weiterschalten
  - Zaehler inkrementieren oder dekrementieren
  - Zeiger auf nächstes Listenelement setzen
  - weiterschalten in Richtung auf eine Beendigung der Schleife
  - Ort: vorzugsweise am Ende des Schleifenkörpers

- Addieren einer Folge von Zahlen mit Endemarke 0

Variante A:

```
int i, Summe;
int Zahl[10] =
     {1,3,7,5,4,9,0,0,0,0};

Summe = 0;
i = 0;
while (Zahl[i] != 0) {
    Summe = Summe + Zahl[i];
    i = i + 1;
}
```

Variante B:

```
int Zahl[10] =
     {1,3,7,5,4,9,0,0,0,0};

int Summe = 0;
int i = 0;
while (Zahl[i]) {
    Summe += Zahl[i++];
}
```

do ... while ... nicht-abweisende Schleife

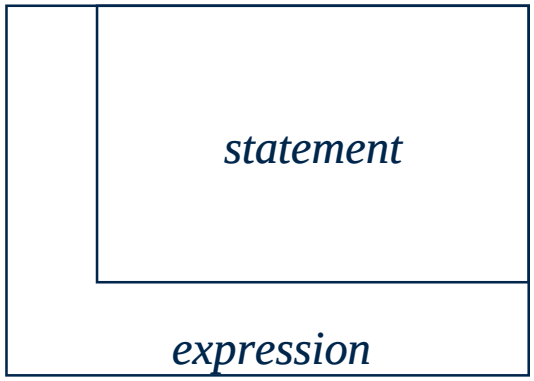
- Bedingung eingeklammert und mit `;` abgeschlossen

- Typ der Bedingung wie bei `if`
- Rücksprung nach `do` wenn *expression* `!=0`
- **1** .. n Schleifendurchläufe

- nur für Sonderfälle
  - im Normalfall `while` verwenden
  - keine Prüfung, ob *statement* überhaupt ausgeführt werden darf

do
statement
while (*expression*);

Laufbedingung



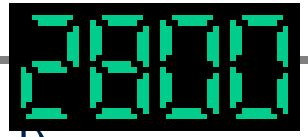
Struktogramm

do ... while ... sinnvoll im Dialog

- Überprüfung eines im Dialog eingegebenen Wertes:

```
const double Minimum = 3;
const double Maximum = 27;
double Wert;
do {
   cout << "Bitte geben Sie einen Wert im Bereich "
        << Minimum << ".." << Maximum << " ein: ";
   cin  >> Wert;
} while (Wert < Minimum || Wert > Maximum);
```

do ... while ... weiteres Beispiel



Zerlegung einer Ganzzahl in Ziffern (BCD: binary coded decimal)

empfehlenswert (Trennung von Ausdrücken und Zuweisungen)

```
int i, Zahl, BCD[16];

i = 0;
do {
    BCD[i] = Zahl % 10;
    Zahl   = Zahl / 10;
    i = i + 1;
} while (Zahl > 0);
```

abzuraten **L**

```
int i, Zahl, BCD[16];

i = 0;
do {
    BCD[i++] = Zahl % 10;
} while (Zahl /= 10);
```

Warum do ... while ? Wenn Zahl = 0, dann wird zumindest eine 0 eingetragen

for ... zählende Schleife

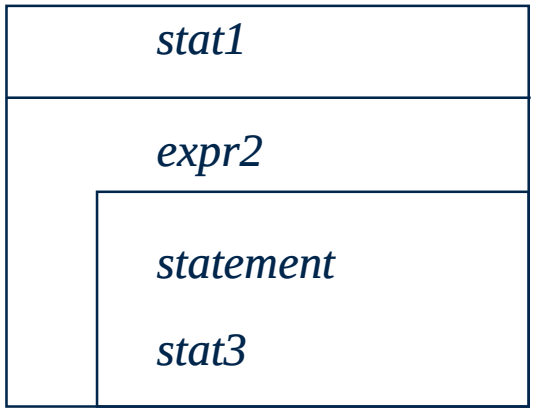
- **mächtig:** nicht auf `int` Zählvariable beschränkt
- *stat1* initialisiert
- *expr2* prüft
- *stat3* inkrementiert
- *statement* wird ausgeführt solange *expr2* `!=0`
- 0 .. n Schleifendurchläufe
- zur Abarbeitung von Arrays empfohlen

Laufbedingung

```
for ( stat1 ; expr2 ; stat3 )
  statement
```

ist äquivalent zu:

```
stat1 ;
while ( expr2 ) {
  statement ;
  stat3 ;
}
```



Struktogramm

Addieren der Elemente eines Arrays:

(Umgang mit `Laenge` bitte einprägen; typisches Schema !)

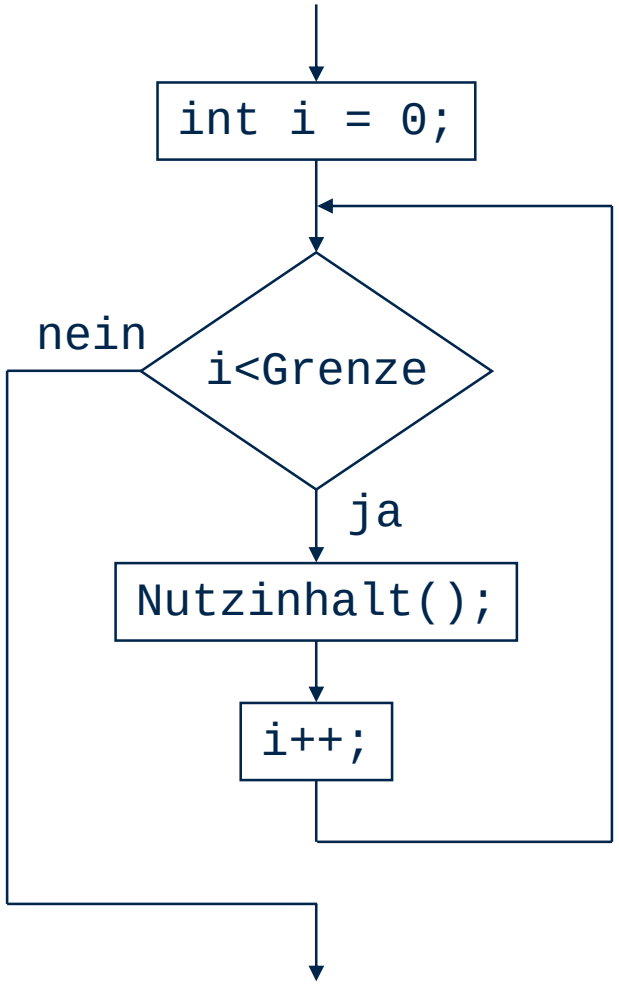
for:

```
const int Laenge = 10;
int Zahl[Laenge] =
     {1,3,7,5,4,9,2,2,5,6};

int Summe = 0;
for (int i=0;i<Laenge;i++){
    Summe += Zahl[i];
}
```

while:

```
const int Laenge = 10;
int Zahl[Laenge] =
     {1,3,7,5,4,9,2,2,5,6};

int Summe = 0;
int i = 0;
while (i<Laenge) {
    Summe += Zahl[i];
    i++;
}
```

Programmablaufplan einer Schleife




```
int i = 0;
while (i<Grenze)
{
   Nutzinhalt();
   i++;
}
```

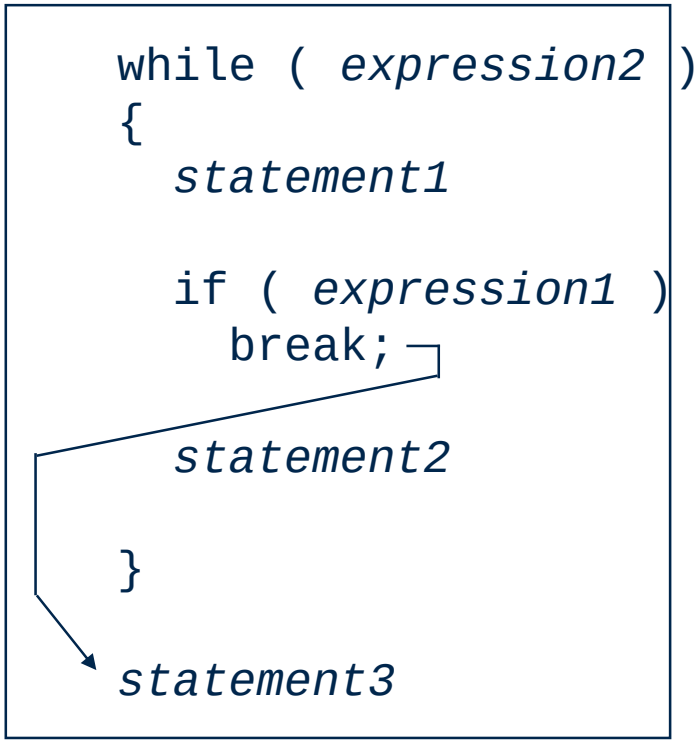
```
for (int i=0;
     i<Grenze;
     i++)
{
   Nutzinhalt();
}
```

for ... und mehrdim. Arrays — Beispiel

```
const int Quartale = 4;                    // Arraygrößen
const int Regionen = 3;
int HessensUmsatz, GesamtUmsatz;           // Akkumulatoren
int Umsatz [Quartale][Regionen] = {
        {00, 01, 02}, {10, 11, 12},        // "Eingabewerte"
        {20, 21, 22}, {30, 31, 32}};

GesamtUmsatz = HessensUmsatz = 0;          // Akkus löschen
for (int q=0; q<Quartale; q++) {           // Schleife über Quartale
    HessensUmsatz += Umsatz [q][1];
    for (int r=0; r<Regionen; r++)         // Schleife über Regionen
        GesamtUmsatz += Umsatz [q][r];
};
```

break vorzeitiger Schleifenausstieg

- notwendig in `switch case`
- verläßt nächstumschließende `for`, `while` oder `do` Schleife
- steht im Widerspruch zur Strukturierten Programmierung
  - läßt sich nicht im Struktogramm abbilden
- normalerweise vermeiden
  - nur in Sonderfällen und dann mit größter Vorsicht einsetzen
  - u.U. vertretbar, wenn *expression2* dadurch an Übersichtlichkeit gewinnt



Beispiel zu break

Primzahlenberechnung

```
for (int Kandidat = Anfang;
     Kandidat <= Ende;
     Kandidat++)
{
  for (int Teiler = 2;
       Teiler < Kandidat;
       Teiler++)
  {
    if (Kandidat%Teiler == 0)
      break;   // keine Primzahl
  }
  if (Teiler < Kandidat)
    // das war Ausstieg über break
    cout << Kandidat << endl;
}
```

mit break: schnell

```
for (int Kandidat = Anfang;
     Kandidat <= Ende;
     Kandidat++)
{
  bool istPrimzahl = true;
  Teiler = 2;
  while (istPrimzahl &&
         (Teiler < Kandidat))
  {
    if (Kandidat%Teiler == 0)
      istPrimzahl = false;
    Teiler++;
  }
  if (istPrimzahl)
    cout << Kandidat << endl;
```

aufwendiger

ohne break: übersichtlich

Richtig?

continue überspringt Schleifenrest

- in `for` Schleifen: springt direkt zur Inkrement-Anweisung
- in `while` oder `do` Schleifen: springt direkt zur Bedingung

```
for ( stat1 ; expr2 ; stat3 ) {
  statement1
  if ( expression1 )
    continue;
  statement2
}
```

- steht im Widerspruch zur Strukturierten Programmierung
- läßt sich nicht im Struktogramm abbilden
- grundsätzlich vermeiden
  - *statement2* läßt sich ohne weiteres unter inverse `if`-Bedingung stellen

```
while ( expression2   ) {
  statement1
  if ( expression1 )
    continue;
  statement2
}
```

goto beliebiger Sprung zu Marke

- `goto` springt ohne Rücksicht auf Schleifen und Schachtelung direkt zur Marke
- Marke hat die Form `name:`
- steht im Widerspruch zur Strukturierten Programmierung
- ruiniert jedes Struktogramm
- ist bei uns **verboten !**

```
j=0;
for (i=0; i<8; i++) {
  j=0;
  while (b[j] > 0) {
    if a[i]==b[j]
      goto gefunden;
    j++;
  }
}
gefunden:
  Ergebnis=b[j];
```

Checkliste beliebter Fehler

q if ... else ...	`else` Zuordnung, Klammerung ?
q switch ... case ...	`break` vorhanden ? `default` vorhanden ?
q while ...	Laufvariable initialisiert ?
q do ... while ...	Inkrement/Dekrement-Anweisung ?
q for ...	Laufbedingung ?
q while, for	kein "`;`" hinter Bedingung bzw.)
q Bedingungen	kein "=" in Bedingung, sondern "=="