Operatoren & Kontrollstrukturen

Parameter	Kursinformationen
Veranstaltung:	`Prozedurale Programmierung / Einführung in die Informatik / Erhebung, Analyse und Visualisierung digitaler Daten`
Semester	`Wintersemester 2025/26`
Hochschule:	`Technische Universität Freiberg`
Inhalte:	`Operatoren / Kontrollstrukturen`
Link auf Repository:	https://github.com/TUBAF-IfI-LiaScript/VL_EAVD/blob/master/02_OperatorenKontrollstrukturen.md
Autoren	Sebastian Zug & André Dietrich & Galina Rudolf & Copilot

Fragen an die heutige Veranstaltung ...

- Wonach lassen sich Operatoren unterscheiden?
- Welche unterschiedliche Bedeutung haben `x++` und `++x`?
- Erläutern Sie den Begriff unärer, binärer und tertiärer Operator.
- Unterscheiden Sie Zuweisung und Anweisung.
- Wie lassen sich Kontrollflüsse grafisch darstellen?
- Welche Konfigurationen erlaubt die `for`-Schleife?
- In welchen Funktionen (Verzweigungen, Schleifen) ist Ihnen das Schlüsselwort `break` bekannt?
- Worin liegt der zentrale Unterschied der `while` und `do-while` Schleife?
- Recherchieren Sie Beispiele, in denen `goto`-Anweisungen Bugs generierten.

Reflexion Ihrer Fragen / Rückmeldungen

> Zur Erinnerung ... Wettstreit zur partizipativen Materialentwicklung mit den Informatikern ...

Preis für das aktivste Auditorium

Format	Informatik Studierende
Verbesserungsvorschlag	0
Fragen	2
generelle Hinweise	0

> Wie verwende ich internationale Sprache in C++?

Achtung: Nicht alle internationalen Zeichen passen in einen einfachen `char` (1 Byte). Beispiel:

```
#include <iostream>
#include <clocale>
int main() {
  std::setlocale(LC_ALL, "de_DE.UTF-8");
  char smiley = '😊'; // funktioniert NICHT wie erwartet!
  std::cout << "char: " << smiley << std::endl;
  wchar_t wsmiley = L'😊'; // funktioniert (sofern Konsole und Compiler Uni
```

```
    unterstützen)
    std::wcout << L"wchar_t: " << wsmiley << std::endl;
    std::cout << "Erledigt : " << std::endl;
    return 0;
}
```

Das Smiley-Symbol benötigt mehr als 1 Byte. Sie haben in einem Byte schlicht und einfach keinen Platz. Darüberhinau sind einzelne Unicode-Zeichen (wie z.B. Emojis) immer etwas "schwierig" in der Handhabung.

Um Umlaute und internationale Zeichen korrekt in C++ zu verwenden, sollte die Quellcodedatei als UTF-8 gespeichert werden. Für die Ausgabe empfiehlt sich die Nutzung von `UTF-8-Strings`, z. B.:

```
#include <iostream>
int main() {
  std::cout << "Grüße aus München! äöü ß ç ñ" << std::endl;    // funktioniert, wenn die Datei als UTF-8 gespeichert ist
  std::cout << u8"Grüße aus München! äöü ß ç ñ" << std::endl;  // expl Vorgabe eines UTF-8-Strings
  return 0;
}
```

```
Grüße aus München! äöü ß ç ñ
Grüße aus München! äöü ß ç ñ
Grüße aus München! äöü ß ç ñ
Grüße aus München! äöü ß ç ñ
```

Worum geht es heute?

Vorgang	Operatoren / Kontrollstrukturen
Durchlaufe die Bestandsliste des Lagers und erhöhe pro Produkt den Lagerbestand um 1, wenn Du es findest.	Schleife bis zum Ende (des Lagerbestandes), Inkrementierung (Addition mit 1)
Berechne die Nullstelle einer Funktion iterativ mit dem Newton-Verfahren. Wiederhole es so lange bis die Abweichung kleiner als ein vorgegebenes Epsilon ist.	Schleife mit Abbruchbedingung, Vergleich von Zahlenwerten, Zuweisung,
Wenn die Temperatur oberhalb von 20 Grad Celsius liegt und es nicht regnet, bewerten wir das Wetter als frühlingshaft, sonst als zu kalt.	Verzweigung mit (zwei !) Bedingungen, Vergleich von Zahlenwerten UND wahr/falsch Aussagen (Boolsche variablen)

Operatoren

Operatoren sind die „Verben“ einer Programmiersprache: Sie charakterisieren "*was passieren soll*" und ermöglichen Berechnungen, Vergleichen, Zuweisen und logische Verknüpfungen.

- `c=a+b`
- `a>b`
- `sizeof(d)`
- ...

1. Arithmetische Operatoren

Merkkasten: Arithmetische Operatoren funktionieren wie in der Mathematik – aber Achtung: Bei Ganzzahlen werden Nachkommastellen einfach abgeschnitten! Das führt oft zu unerwarteten Ergebnissen.

Operator	Bedeutung	Beispiel (C++)	Alltagsbeispiel
`+`	Addition	`a + b`	
`-`	Subtraktion	`a - b`	
`*`	Multiplikation	`a * b`	
`/`	Division	`a / b`	20 Seiten / 4 Tage = 5 Seiten pro Tag
`%`	Modulo (Rest)	`a % b`	20 Seiten % 3 Tage = 2 Seiten bleiben übrig

Typische Fehlerquelle: Bei Ganzzahlen liefert `/` nur den ganzzahligen Anteil, `%` den Rest. Beispiel: `7 / 3 = 2`, `7 % 3 = 1`.

- Wenn zwei Ganzzahlen wie z. B. $4/3$ dividiert werden, erhalten wir das Ergebnis 1 zurück, der nicht ganzzahlige Anteil der Lösung bleibt unbeachtet.
- Für Fließkommazahlen wird die Division wie erwartet realisiert.

division.cpp

```cpp
#include <iostream>

int main(){
  int a=20;
  float b=20.f;

  std::cout << "Ganzzahlige Division 20 / 3 = " << a/3 << "\n";
  std::cout << "Gleitkomma Division 20 / 3 = " << b/3 << "\n";
  return 0;
}
```

```
Ganzzahlige Division 20 / 3 = 6
Gleitkomma Division 20 / 3 = 6.66667
Ganzzahlige Division 20 / 3 = 6
Gleitkomma Division 20 / 3 = 6.66667
```

moduloExample.cpp

```cpp
#include <iostream>

int main(){
  int timestamp, sekunden, minuten;

  timestamp = 345; //[s]
  std::cout << "Zeitstempel " << timestamp << " [s]\n";
  minuten=timestamp/60;
  sekunden=timestamp%60;
  std::cout << "Besser lesbar: " << minuten << " min. " << sekunden <
    sek.\n";
  return 0;
}
```

```
Zeitstempel 345 [s]
Besser lesbar: 5 min. 45 sek.
Zeitstempel 345 [s]
Besser lesbar: 5 min. 45 sek.
```

2. Vergleichsoperatoren

Warnung: Verwechsle niemals das Gleichheitszeichen `==` (Vergleich) mit dem Zuweisungszeichen `=`! Das ist einer der häufigsten Anfängerfehler und kann schwer auffindbare Bugs verursachen.

Kern der Logik sind Aussagen, die wahr oder falsch sein können.

Operator	Bedeutung	Beispiel (C++)	Alltagsbeispiel
`==`	Gleich	`a == b`	Ist die Temperatur genau 20°C?
`!=`	Ungleich	`a != b`	Ist die Note nicht 1,0?
`<`	Kleiner	`a < b`	Ist das Alter < 18?
`<=`	Kleiner/gleich	`a <= b`	Ist die Geschwindigkeit ≤ 50 km/h?
`>`	Größer	`a > b`	Ist der Kontostand > 0?
`>=`	Größer/gleich	`a >= b`	Ist die Temperatur ≥ 100°C?

Typische Fehlerquelle: Verwechsle nie `=` (Zuweisung) und `==` (Vergleich)!

LogicOperators.cpp

```cpp
#include <iostream>

int main(){
  int x = 15;
  std::cout << "x = " << x << " \n";
  std::cout << std::boolalpha << "Aussage x > 5 ist "<< (x>5) << "\n";
  std::cout << std::boolalpha << "Aussage x == 5 ist "<< (x==-15) << "
  return 0;
}
```

```
x = 15
Aussage x > 5 ist true
Aussage x == 5 ist false
```

Merke: Der Rückgabewert einer Vergleichsoperation ist `bool`. Dabei bedeutet `false` eine ungültige und `true` eine gültige Aussage.

Vor 1993 wurde ein logischer Datentyp in C++ durch `int` simuliert. Aus der Gründen der Kompatibilität wird `bool` überall, wo wie hier nicht ausdrücklich `bool` verlangt wird in `int` (Werte `0` und `1`) umgewandelt.

Mit dem `boolalpha` Parameter kann man `cout` überreden zumindest `true` und `false` auszugeben.

3. Logische Operatoren

Praxis-Tipp: Kombiniere Bedingungen mit logischen Operatoren, um komplexe Prüfungen in einer einzigen if-Abfrage auszudrücken. Das macht deinen Code übersichtlicher und vermeidet unnötige Verschachtelungen.

Und wie lassen sich logische Aussagen verknüpfen? Nehmen wir an, dass wir aus den Messdaten zweier Sensoren ein Alarmsignal generieren wollen. Nur wenn die Temperatur *und* die Luftfeuchte in einem bestimmten Fenster liegen, soll dies nicht passieren.

Operator	Bedeutung	Beispiel (C++)	Alltagsbeispiel
`&&`	UND	`a && b`	„Ich gehe raus, wenn es warm **und** trocken ist.“
`||`	ODER	`a || b`	„Ich nehme Tee **oder** Kaffee.“
`!`	NICHT	`!a`	„Ich gehe **nicht** schwimmen.“

Praxis-Tipp: Logische Operatoren verknüpfen Bedingungen, z.B. in if-Abfragen oder Schleifen.

Das ODER wird durch senkrechte Striche repräsentiert (Altgr+`<` Taste) und nicht durch große `I`!

Nehmen wir an, sie wollen Messdaten evaluieren. Ihr Sensor funktioniert nur dann wenn die Temperatur ein Wert zwischen -10 und -20 Grad annimmt und die Luftfeuchte zwischen 40 bis 60 Prozent beträgt.

Logic.cpp

```cpp
#include <iostream>

int main(){
  float Temperatur = -30;     // Das sind unsere Probewerte
  float Feuchte = 65;

  // Vergleichsoperationen und Logische Operationen
  bool Temp_valid = ....    // Hier sind Sie gefragt!
  bool Feuchte_valid = ....

  // Ausgabe
  if ... {
    std::cout << "Die Messwerte kannst Du vergessen!\n";
  }else{
    std::cout << "Die Messwerte sind in Ordnung.\n";
  }
  return 0;
}
```

```
main.cpp: In function ‘int main()’:
main.cpp:8:21: error: expected primary-expression before ‘...’ token
    8 |   bool Temp_valid = ....    // Hier sind Sie gefragt!
      |                     ^~~
main.cpp:4:9: warning: unused variable ‘Temperatur’ [-Wunused-variable]
    4 |   float Temperatur = -30;     // Das sind unsere Probewerte
      |         ^~~~~~~~~~
main.cpp:5:9: warning: unused variable ‘Feuchte’ [-Wunused-variable]
    5 |   float Feuchte = 65;
      |         ^~~~~~~
main.cpp:8:8: warning: unused variable ‘Temp_valid’ [-Wunused-variable]
    8 |   bool Temp_valid = ....    // Hier sind Sie gefragt!
      |        ^~~~~~~~~~
main.cpp: In function ‘int main()’:
main.cpp:8:21: error: expected primary-expression before ‘...’ token
    8 |   bool Temp_valid = ....    // Hier sind Sie gefragt!
      |                     ^~~
main.cpp:4:9: warning: unused variable ‘Temperatur’ [-Wunused-variable]
    4 |   float Temperatur = -30;     // Das sind unsere Probewerte
      |         ^~~~~~~~~~
main.cpp:5:9: warning: unused variable ‘Feuchte’ [-Wunused-variable]
    5 |   float Feuchte = 65;
      |         ^~~~~~~
main.cpp:8:8: warning: unused variable ‘Temp_valid’ [-Wunused-variable]
    8 |   bool Temp_valid = ....    // Hier sind Sie gefragt!
      |        ^~~~~~~~~~
```

Anmerkung: C++ bietet für logische Operatoren und Bit-Operatoren Synonyme `and`, `or`, `xor`. Die Synonyme sind Schlüsselwörter, die die Lesbarkeit deutlich erhöhen.

```cpp
#include <iostream>
int main() {
  float Temperatur = -30;
  float Feuchte = 65;

  bool Temp_valid = Temperatur > -20 and Temperatur < -10;
  bool Feuchte_valid = Feuchte >= 40 and Feuchte <= 60;

  if (not (Temp_valid and Feuchte_valid)) {
    std::cout << "Die Messwerte kannst Du vergessen!\n";
  } else {
    std::cout << "Die Messwerte sind in Ordnung.\n";
  }
```

```
    }
    return 0;
}
```

```
Die Messwerte kannst Du vergessen!
```

Hinweis: Die Synonyme `and`, `or`, `not` können die Lesbarkeit erhöhen, sind aber nicht in allen C++-Compilern und -Einstellungen standardmäßig aktiviert.

4. Zuweisungsoperatoren

Der Zuweisungsoperator `=` ist von seiner mathematischen Bedeutung zu trennen -

einer Variablen wird ein Wert zugeordnet. Damit ergibt dann auch `x=x+1` Sinn.

Operator	Bedeutung	Beispiel (C++)	Alltagsbeispiel
`=`	Zuweisung	`a = 5`	„Lege fest: Kontostand = 100 €“
`+=`	Addition & Zuweisung	`a += 2`	„Kontostand um 2 € erhöhen“
`-=`	Subtraktion & Zuweisung	`a -= 1`	„1 € abheben“

Merke: Zuweisung ist kein Vergleich! `a = 5` setzt den Wert, `a == 5` prüft ihn.

zuweisung.cpp

```cpp
#include <iostream>

int main() {
    int zahl1 = 10;
    int zahl2 = 20;
    int ergeb = 0;
    // Zuweisung des Ausdrucks 'zahl1 + zahl2'
    ergeb = zahl1 + zahl2;

    std::cout << zahl1 << " + " << zahl2 << " = " << ergeb << "\n";
    return 0;
}
```

```
10 + 20 = 30
10 + 20 = 30
```

Achtung: Verwechseln Sie nicht den Zuweisungsoperator `=` mit dem Vergleichsoperator `==`. Der Compiler kann die Fehlerhaftigkeit kaum erkennen und generiert Code, der ein entsprechendes Fehlverhalten zeigt.

Besondere Operatoren

Operatoren lassen sich nach der Anzahl der Operanden klassifizieren:

Operatorenart	Beschreibung	Beispiele
Unär	Ein Operand (z.B. Inkrement, Negation, logische Negation)	`++x`, `-x`, `!x`
Binär	Zwei Operanden (z.B. Addition, Subtraktion, Multiplikation, logisches UND)	`a + b`, `a - b`, `a * b`, `a && b`
Ternär	Drei Operanden (z.B. Bedingungsoperator)	`a ? b : c`, `x > 0 ? x : -x`

Inkrement und Dekrement

Mit den `++` und `--` -Operatoren kann ein Wert um eins erhöht bzw. um eins vermindert werden. Man kan dies vor odr nach der Verarbeitung umsetzen.

IncrementDecrement.cpp

```cpp
#include <iostream>

int main(){
  int x, result;
  x = 5;
  result = 2 * ++x;   // Gebrauch als Präfix
  std::cout << "x=" << x << " und result=" << result << "\n";
  result = 2 * x++;   // Gebrauch als Postfix
  std::cout << "x=" << x << " und result=" << result << "\n";
  return 0;
}
```

```
x=6 und result=12
x=7 und result=12
x=6 und result=12
x=7 und result=12
```

`sizeof` - Operator

`sizeof` ist ein Operator, der bereits beim Übersetzen des Programms (compile time) ausgewertet wird. Er liefert die Größe eines Datentyps oder einer Variable in Bytes – und zwar **ohne** dass das Programm dafür laufen muss. Das ist effizient, sicher und ermöglicht viele typische Aufgaben in C/C++:

- `sizeof` ist keine Funktion, sondern ein Operator.
- `sizeof` wird häufig zur dynamischen Speicherreservierung verwendet.

sizeof.cpp

```cpp
#include <iostream>

int main(){
  double wert=0.0;
  std::cout << sizeof(0) << " " << sizeof(double) << " " << sizeof(wer
    "\n";
  return 0;
}
```

```
4 8 8
4 8 8
```

Vorrangregeln

Konsequenterweise bildet auch die Programmiersprache C/C++ eigene Vorrangregeln ab, die grundlegende mathematische Definitionen "Punktrechnung vor Strichrechnung" realisieren. Die Liste der unterschiedlichen Operatoren macht aber weitere Festlegungen notwendig.

Prioritäten

In welcher Reihung erfolgt beispielsweise die Abarbeitung des folgenden Ausdruckes?

```
c = sizeof(x) + ++a / 3;
```

Für jeden Operator wurde eine Priorität definiert, die die Reihung der Ausführung regelt.

Liste der Vorrangregeln

Im Beispiel bedeutet dies:

```
c = sizeof(x) + ++a  / 3;
//    |       |  |   |
//    |       |  |   |--- Priorität 13
//    |       |  |--- Priorität 14
//    |       |--- Priorität 12
//    |--- Priorität 14

c = (sizeof(x)) + ((++a) / 3);
```

Für Operatoren mit der gleichen Priorität ist für die Reihenfolge der Auswertung die Assoziativität das zweite Kriterium.

sizeof.cpp

```
1  #include <iostream>
2
3  int main(){
4  // von rechts nach links (FALSCH)
5  // 4 / (2 / 2)   // ergibt 4
6
7  // von links nach rechts ausgewertet
8  // (4 / 2) / 2   // ergibt 1
9    double a = 4 / 2 / 2;
10   std::cout << a << "\n";
11   return 0;
12 }
```

```
1
1
```

Merke: Setzen Sie Klammern, um alle Zweifel auszuräumen!

... und mal praktisch

Folgender Code nutzt die heute besprochenen Operatoren um die Eingaben von zwei Buttons auf eine LED abzubilden. Nur wenn beide Taster gedrückt werden, beleuchte das rote Licht für 3 Sekunden.

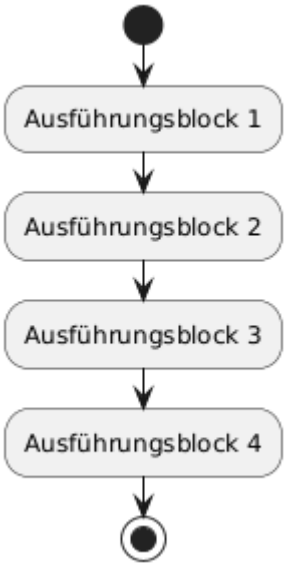
Wie ändert sich die Logik wenn Sie ein `||` anstelle des `&&` verwenden?

ButtonSynch.ino

```
const int button1Pin = A0;
const int button2Pin = A1;
const int ledPin = 8;

void setup() {
  pinMode(button1Pin, INPUT_PULLUP);
  pinMode(button2Pin, INPUT_PULLUP);
  pinMode(ledPin, OUTPUT);
}

void loop() {
  // Taster sind aktiv LOW (wegen INPUT_PULLUP)
  bool button1Pressed = digitalRead(button1Pin) == HIGH;
  bool button2Pressed = digitalRead(button2Pin) == HIGH;

  if (button1Pressed || button2Pressed) {
    digitalWrite(ledPin, HIGH); // LED an
  } else {
    digitalWrite(ledPin, LOW);  // LED aus
  }
}
```

Kontrollfluss

Bisher haben wir Programme entworfen, die eine sequenzielle Abfolge von Anweisungen enthielt.



Lineare Ausführungskette

Diese Einschränkung wollen wir nun mit Hilfe weiterer Anweisungen überwinden:

1. **Verzweigungen (Selektion)**: In Abhängigkeit von einer Bedingung wird der Programmfluss an unterschiedlichen Stellen fortgesetzt.

   Beispiel: Wenn bei einer Flächenberechnung ein Ergebnis kleiner Null generiert wird, erfolgt eine Fehlerausgabe. Sonst wird im Programm fortgefahren.

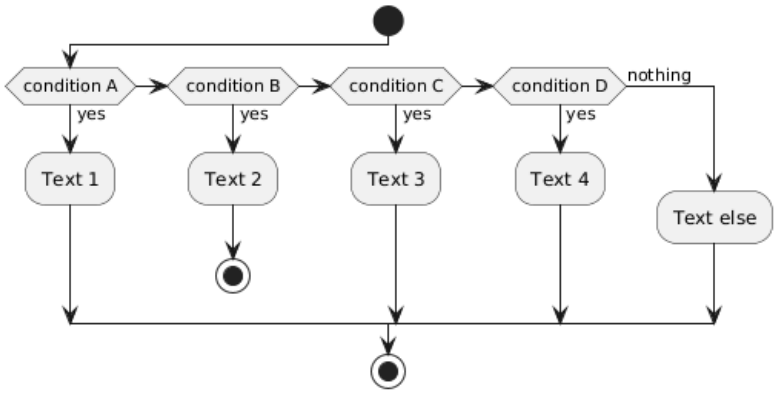
2. **Schleifen (Iteration)**: Ein Anweisungsblock wird so oft wiederholt, bis eine Abbruchbedingung erfüllt wird.

   Beispiel: Ein Datensatz wird durchlaufen um die Gesamtsumme einer Spalte zu bestimmen. Wenn der letzte Eintrag erreicht ist, wird der Durchlauf abgebrochen und das Ergebnis ausgegeben.

3. Des Weiteren verfügt C/C++ über **Sprünge**: die Programmausführung wird mit Hilfe von Sprungmarken an einer anderen Position fortgesetzt. Formal sind sie jedoch nicht notwendig. Statt die nächste Anweisung auszuführen, wird (zunächst) an eine ganz andere Stelle im Code gesprungen.

Verzweigungen

Verzweigungen entfalten mehrere mögliche Pfade für die Ausführung des Programms.



Darstellungsbeispiele für mehrstufige Verzweigungen (`switch`)

`if`-Anweisungen

Im einfachsten Fall enthält die `if`-Anweisung eine einzelne bedingte Anweisung oder einen Anweisungsblock. Sie kann mit `else` um eine Alternative erweitert werden.

Zum Anweisungsblock werden die Anweisungen mit geschweiften Klammern (`{` und `}`) zusammengefasst.

```
if (Bedingung) Anweisung;   // <- Einzelne Anweisung

if (Bedingung) {            // <- Beginn Anweisungsblock
  Anweisung;
  Anweisung;
}                           // <- Ende Anweisungsblock
```

Optional kann eine alternative Anweisung angegeben werden, wenn die Bedingung nicht erfüllt wird:

```
if (Bedingung) {
  Anweisung;
} else {
  Anweisung;
}
```

Mehrere Fälle können verschachtelt abgefragt werden:

```
if (Bedingung)
  Anweisung;
else
  if (Bedingung)
    Anweisung;
  else
    Anweisung;
    Anweisung;      //!!!
```

Merke: An diesem Beispiel wird deutlich, dass die Klammern für die Zuordnung elementar wichtig sind. Die letzte Anweisung gehört NICHT zum zweiten `else` Zweig und auch nicht zum ersten. Diese Anweisung wird immer ausgeführt! Der Kompiler kann dieses Verhalten finden und mit der Option `-Wmisleading-indentation` aufzeigen.

Weitere Beispiele für Bedingungen

Die Bedingungen können als logische UND arithmetische Ausdrücke formuliert werden.

Ausdruck	Bedeutung
`if (a != 0)`	$a \neq 0$
`if (a == 0)`	$a = 0$
`if (!(a <= b))`	$\overline{(a \leq b)}$ oder $a > b$
`if (a != b)`	$a \neq b$
`if (a || b)`	$a > 0$ oder $b > 0$

Mögliche Fehlerquellen

1. Zuweisungs- statt Vergleichsoperator in der Bedingung (kein Compilerfehler)
2. Bedingung ohne Klammern (Compilerfehler)
3. `;` hinter der Bedingung (kein Compilerfehler)
4. Multiple Anweisungen ohne Anweisungsblock
5. Komplexität der Statements

errors.cpp

```cpp
#include <iostream>

int main(){
  int a = 5, b = 10;
  if (a = b) {                        // Fehler 1
    std::cout << "a gleich b\n";
  }
  if a != b {                         // Fehler 2
    std::cout << "a ungleich b\n";
  }
  if (a < b); {                       // Fehler 3
    std::cout << "a kleiner b\n";
  }
  if (a > b)                          // Fehler 4
    std::cout << "a größer b\n";
    std::cout << "Ende\n";
  if ((a < b) && (b > 0) || (a == 5)) { // Fehler 5
    std::cout << "Komplexe Bedingung\n";
  }
  return 0;
}
```

```
main.cpp: In function ‘int main()’:
main.cpp:5:9: warning: suggest parentheses around assignment used as
truth value [-Wparentheses]
    5 |   if (a = b) {                     // Fehler 1
      |       ~~^~~
main.cpp:8:6: error: expected ‘(’ before ‘a’
    8 |   if a != b {                      // Fehler 2
      |      ^
      |      (
main.cpp:11:3: warning: this ‘if’ clause does not guard... [-
Wmisleading-indentation]
   11 |   if (a < b); {                    // Fehler 3
      |   ^~
main.cpp:11:15: note: ...this statement, but the latter is misleadingly
indented as if it were guarded by the ‘if’
   11 |   if (a < b); {                    // Fehler 3
      |               ^
main.cpp:14:3: warning: this ‘if’ clause does not guard... [-
Wmisleading-indentation]
   14 |   if (a > b)                       // Fehler 4
      |   ^~
main.cpp:16:5: note: ...this statement, but the latter is misleadingly
indented as if it were guarded by the ‘if’
   16 |     std::cout << "Ende\n";
      |     ^~~
main.cpp:17:15: warning: suggest parentheses around ‘&&’ within ‘||’ [-
Wparentheses]
   17 |   if ((a < b) && (b > 0) || (a == 5)) { // Fehler 5
      |       ~~~~~~~~^~~~~~~~~~
main.cpp: In function ‘int main()’:
main.cpp:5:9: warning: suggest parentheses around assignment used as
truth value [-Wparentheses]
    5 |   if (a = b) {                     // Fehler 1
      |       ~~^~~
main.cpp:8:6: error: expected ‘(’ before ‘a’
    8 |   if a != b {                      // Fehler 2
      |      ^
      |      (
main.cpp:11:3: warning: this ‘if’ clause does not guard... [-
Wmisleading-indentation]
   11 |   if (a < b); {                    // Fehler 3
      |   ^~
main.cpp:11:15: note: ...this statement, but the latter is misleadingly
```

```
indented as if it were guarded by the ‘if’
   11 |   if (a < b); {                    // Fehler 3
      |               ^
main.cpp:14:3: warning: this ‘if’ clause does not guard... [-
Wmisleading-indentation]
   14 |   if (a > b)                       // Fehler 4
      |   ^~
main.cpp:16:5: note: ...this statement, but the latter is misleadingly
indented as if it were guarded by the ‘if’
   16 |     std::cout << "Ende\n";
      |     ^~~
main.cpp:17:15: warning: suggest parentheses around ‘&&’ within ‘||’ [-
Wparentheses]
   17 |   if ((a < b) && (b > 0) || (a == 5)) { // Fehler 5
      |       ~~~~~~~~^~~~~~~~~~
```

Zwischenfrage

Test.cpp

```cpp
#include <iostream>

int main() {
  int Punkte = 45;
  int Zusatzpunkte = 15;
  if (Punkte + Zusatzpunkte >= 50)
  {
    std::cout << "Test ist bestanden!\n";
    if (Zusatzpunkte < 15)
    {
      if(Zusatzpunkte > 8) {
        std::cout << "Respektable Leistung\n";
      }
    }else{
      std::cout << "Alle Zusatzpunkte geholt!\n";
    }
  }else{
    std::cout << "Leider durchgefallen!\n";
  }
  return 0;
}
```

```
Test ist bestanden!
Alle Zusatzpunkte geholt!
Test ist bestanden!
Alle Zusatzpunkte geholt!
```

- `Test ist bestanden`
- `Alle Zusatzpunkte geholt`
- `Leider durchgefallen!`
- `Test ist bestanden!+Alle Zusatzpunkte geholt!`
- `Test ist bestanden!+Respektable Leistung`

`switch`-Anweisungen

> *Too many ifs - I think I switch*
>
> Berndt Wischnewski

Eine übersichtlichere Art der Verzweigung für viele, sich ausschließende Bedingungen wird durch die `switch`-Anweisung bereitgestellt. Sie wird in der Regel verwendet, wenn eine oder einige unter vielen Bedingungen ausgewählt werden sollen. Das Ergebnis der "expression"-Auswertung soll eine Ganzzahl (oder `char`-Wert) sein. Stimmt es mit einem "const_expr"-Wert überein, wird die Ausführung an dem entsprechenden `case`-Zweig fortgesetzt. Trifft keine der Bedingungen zu, wird der `default`-Fall aktiviert.

```
switch(expression) {
  case const-expr: Anweisung break;
  case const-expr:
    Anweisungen
    break;
  case const-expr: Anweisungen break;
  default: Anweisungen
}
```

SwitchExample.cpp

```cpp
#include <iostream>

int main() {
  int a=50, b=60;
  char op;
  std::cout << "Bitte Operator definieren (+,-,*,/): ";
  std::cin >> op;

  switch(op) {
    case '+':
      std::cout << a << " + " << b << " = " << a+b << " \n";
      break;
    case '-':
      std::cout << a << " - " << b << " = " << a-b << " \n";
      break;
    case '*':
      std::cout << a << " * " << b << " = " << a*b << " \n";
      break;
    case '/':
      std::cout << a << " / " << b << " = " << a/b << " \n";
      break;
    default:
      std::cout << op << "? kein Rechenoperator \n";
  }
  return 0;
}
```

```
Bitte Operator definieren (+,-,*,/): Bitte Operator definieren
(+,-,*,/):
```

Im Unterschied zu einer `if`-Abfrage wird in den unterschiedlichen Fällen immer nur auf Gleichheit geprüft! Eine abgefragte Konstante darf zudem nur einmal abgefragt werden und muss ganzzahlig oder `char` sein.

```cpp
// Fehlerhafte case Blöcke
int x = 100;
switch(x) {
  case x < 100: // das ist ein Fehler
    y = 1000;
    break;

  case 100.1: // das ist genauso falsch
    y = 5000;
    z = 3000;
    break;
}
```

Und wozu brauche ich das `break`? Ohne das `break` am Ende eines Falls werden alle darauf folgenden Fälle bis zum Ende des `switch` oder dem nächsten `break` zwingend ausgeführt.

SwitchBreak.cpp

```cpp
#include <iostream>

int main() {
  int a=5;

  switch(a) {
    case 5:   // Multiple Konstanten
    case 6:
    case 7:
      std::cout << "Der Wert liegt zwischen 4 und 8\n";
    case 3:
      std::cout << "Der Wert ist 3 \n";
      break;
    case 0:
      std::cout << "Der Wert ist 0 \n";
    default: std::cout << "Wert in keiner Kategorie\n";
  }

  return 0;
}
```

Unter Ausnutzung von `break` können Kategorien definiert werden, die aufeinander aufbauen und dann übergreifend "aktiviert" werden.

CharClassification.cpp

```cpp
#include <iostream>

int main() {
  char ch;
  std::cout << "Geben Sie ein Zeichen ein:  ";
  std::cin >> ch;

  switch(ch)
  {
    case 'a':
    case 'A':
    case 'e':
    case 'E':
    case 'i':
    case 'I':
    case 'o':
    case 'O':
    case 'u':
    case 'U':
      std::cout << ch << " ist ein Vokal.\n";
      break;
    default:
      std::cout << ch << " ist ein Konsonant.\n";
  }
  return 0;
}
```

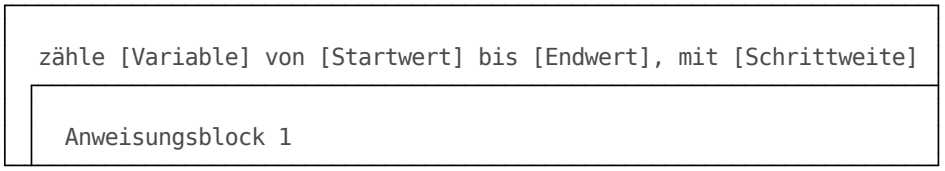
```
Geben Sie ein Zeichen ein:  Geben Sie ein Zeichen ein:
```

Schleifen

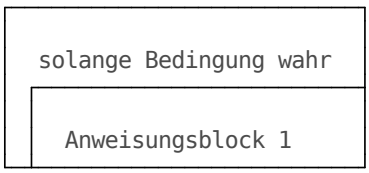
Schleifen dienen der Wiederholung von Anweisungsblöcken – dem sogenannten Schleifenrumpf oder Schleifenkörper – solange die Schleifenbedingung als Laufbedingung gültig bleibt bzw. als Abbruchbedingung nicht eintritt. Schleifen, deren Schleifenbedingung immer zur Fortsetzung führt oder die keine Schleifenbedingung haben, sind *Endlosschleifen*.

Schleifen können verschachtelt werden, d.h. innerhalb eines Schleifenkörpers können weitere Schleifen erzeugt und ausgeführt werden. Zur Beschleunigung des Programmablaufs werden Schleifen oft durch den Compiler entrollt (*Enrollment*).

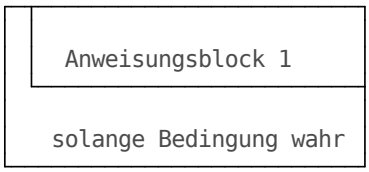
Grafisch lassen sich die wichtigsten Formen in mit der Nassi-Shneiderman Diagrammen wie folgt darstellen:



- Wiederholungsstruktur mit vorausgehender Bedingungsprüfung



- Wiederholungsstruktur mit nachfolgender Bedingungsprüfung



Die Programmiersprache C/C++ kennt diese drei Formen über die Schleifenkonstrukte `for`, `while` und `do while`.

`for`-Schleife

Der Parametersatz der `for`-Schleife besteht aus zwei Anweisungsblöcken und einer Bedingung, die durch Semikolons getrennt werden. Mit diesen wird ein **Schleifenzähler** initiiert, dessen Manipulation spezifiziert und das Abbruchkriterium festgelegt. Häufig wird die Variable mit jedem Durchgang inkrementiert oder dekrementiert, um dann anhand eines Ausdrucks evaluiert zu werden. Es wird überprüft, ob die Schleife fortgesetzt oder abgebrochen werden soll. Letzterer Fall tritt ein, wenn dieser den Wert false (falsch) annimmt.

```
// generisches Format der for-Schleife
for (Initialisierung; Bedingung; Reinitialisierung) {
    // Anweisungen
}

// for-Schleife als Endlosschleife
for (;;) {
    // Anweisungen
}
```

ForLoopExample.cpp

```cpp
#include <iostream>

int main(){
  int i;
  for (i = 1; i<10; i++)
    std::cout << i << " ";

  std::cout << "\nNach der Schleife hat i den Wert " << i << "\n";
  return 0;
}
```

```
1 2 3 4 5 6 7 8 9
Nach der Schleife hat i den Wert 10
1 2 3 4 5 6 7 8 9
Nach der Schleife hat i den Wert 10
```

Beliebte Fehlerquellen

- Semikolon hinter der schließenden Klammer von `for`
- Kommas anstatt Semikolons zwischen den Parametern von `for`
- fehlerhafte Konfiguration von Zählschleifen
- Nichtberücksichtigung der Tatsache, dass die Zählvariable nach dem Ende der Schleife über dem Abbruchkriterium liegt

SemicolonAfterFor.cpp

```cpp
#include <iostream>

int main(){
  int i;
  for (i = 1; i<10; i++);
    std::cout << i << " ";

  std::cout << "Das ging jetzt aber sehr schnell ... Warum eigentlich
    << i;
  return 0;
}
```

```
10 Das ging jetzt aber sehr schnell ... Warum eigentlich?
1010 Das ging jetzt aber sehr schnell ... Warum eigentlich?
10
```

`while`-Schleife

Während bei der `for`-Schleife auf ein n-maliges Durchlaufen Anweisungsfolge konfiguriert wird, definiert die `while`-Schleife nur eine Bedingung für den Fortführung/Abbruch.

```
// generisches Format der while-Schleife
while (Bedingung)
  Anweisungen;

while (Bedingung) {
  Anweisungen;
  Anweisungen;
}
```

count_plus.cpp

```
#include <iostream>

int main() {
  char c;
  int zaehler = 0;
  std::cout << "Pluszeichenzähler - zum Beenden \"_\" [Enter]\n";
  std::cin >> c;
  while (c != '_')
  {
    if (c == '+')
      zaehler++;
    std::cin >> c;
  }
  std::cout << "Anzahl der Pluszeichen: " << zaehler << "\n";
  return 0;
}
```

```
Pluszeichenzähler - zum Beenden "_" [Enter]
Pluszeichenzähler - zum Beenden "_" [Enter]
```

Dabei soll erwähnt werden, dass eine `while`-Schleife eine `for`-Schleife ersetzen kann.

```
// generisches Format der while-Schleife
i = 0;
while (i<10) {
  // Anweisungen;
  i++;
}

for (i=0; i<10; i++) {
  // Anweisungen;
```

```
}
```

`do-while`-Schleife

Im Gegensatz zur `while`-Schleife führt die `do-while`-Schleife die Überprüfung des Abbruchkriteriums erst am Schleifenende aus.

```
// generisches Format der while-Schleife
do
  Anweisung;
while (Bedingung);
```

Welche Konsequenz hat das? Die `do-while`-Schleife wird in jedem Fall einmal ausgeführt.

count_plus.cpp

```
1  #include <iostream>
2
3  int main(){
4    char c;
5    int zaehler = 0;
6    std::cout << "Pluszeichenzähler - zum Beenden \"_\" [Enter]\n";
7    do {
8      std::cin >> c;
9      if(c == '+')
10       zaehler++;
11   } while(c != '_');
12   std::cout << "Anzahl der Pluszeichen: " << zaehler << "\n";
13   return 0;
14 }
```

```
Pluszeichenzähler - zum Beenden "_" [Enter]
Pluszeichenzähler - zum Beenden "_" [Enter]
```

Kontrolliertes Verlassen der Anweisungen

Bei allen drei Arten der Schleifen kann zum vorzeitigen Verlassen der Schleife `break` benutzt werden. Damit wird aber nur die unmittelbar umgebende Schleife beendet!

breakForLoop.cpp

```cpp
#include <iostream>

int main() {
  int i;
  for (i = 1; i<10; i++) {
    if (i == 5) break;
    std::cout << i << " ";
  }
  std::cout << "\nUnd vorbei ... i ist jetzt " << i << "\n";
  return 0;
}
```

```
1 2 3 4
Und vorbei ... i ist jetzt 5
```

Eine weitere wichtige Eingriffsmöglichkeit für Schleifenkonstrukte bietet `continue`. Damit wird nicht die Schleife insgesamt, sondern nur der aktuelle Durchgang gestoppt.

continueForLoop.cpp

```cpp
#include <iostream>

int main() {
  int i;
  for (i = -5; i<6; i++) {
    if (i == 0) continue;
    std::cout << 12.0 / i << "\n";
  }
  return 0;
}
```

```
-2.4
-3
-4
-6
-12
12
6
4
3
2.4
```

Durch `return` - Anweisung wird das Verlassen einer Funktion veranlasst (genaues in der Vorlesung zu Funktionen).

Beispiel des Tages

Divisors.cpp

```cpp
// A Better (than Naive) Solution to find all divisors
#include <iostream>
#include <math.h>

int main() {
  int n = 100;
  std::cout << "The divisors of " << n << " are:\n";

  // Die naive Lösung
  for (int i = 1; i <= n; i++)
    if (n % i == 0)
      std::cout << " " << i << "\n";
  return 0;
}
```

```
The divisors of 100 are:
 1
 2
 4
 5
 10
 20
 25
 50
 100
```

Aufgabe: Wie können wir die Laufzeit des Codes verbessern?

ImprovedDivisors.cpp

```cpp
// A Better (than Naive) Solution to find all divisors
#include <iostream>
#include <cmath>


int main()
{
  int n = 100;
  std::cout << "The divisors of " << n << " are:\n";

  // Diesmal laufen wir nur bis zu Wurzel von n
  for (int i=1; i <= std::sqrt(n); i++)
  {
    if (n % i == 0)
    {
      // Wenn die beiden Teiler gleich sind, dann nur einen ausgeben
      if (n/i == i)
        std::cout << " " << i << "\n";
      else // sonst alle beide
        std::cout << " " << i << " " << n/i << "\n";
    }
  }
  return 0;
}
```

```
The divisors of 100 are:
 1 100
 2 50
 4 25
 5 20
 10
```

Quizze

Operatoren

Operatorentypen

Ordnen Sie die Operatoren den richtigen Bezeichnungen zu.

Unär	Binär	Ternär	
◯	◯	◯	`-` in der Anweisung `b=-a;`
◯	◯	◯	`-` in der Anweisung `b=a-1;`
◯	◯	◯	`sizeof()`
◯	◯	◯	`?`
◯	◯	◯	`+`
◯	◯	◯	`%`

Ordnen Sie die Operatoren den richtigen Bezeichnungen zu.

Infix	Präfix	Postfix	
◯	◯	◯	`a=b+c;`
◯	◯	◯	`a=++b;`
◯	◯	◯	`a=b++;`
◯	◯	◯	`a=a%3;`
◯	◯	◯	`a=&b;`

Zuweisungs- und Vergleichsoperatoren

Ordnen Sie die Operatoren den richtigen Bezeichnungen zu.

Zuweisungsoperator	Vergleichsoperator	
◯	◯	`>=`
◯	◯	`<=`
◯	◯	`==`
◯	◯	`<`
◯	◯	`>`
◯	◯	`=`
◯	◯	`!=`

Inkrement und Dekrement

Verkürzen Sie `x=x+1;` möglichst weit.

Verkürzen Sie `x=x-1;` möglichst weit.

Arithmetische Operatoren

Welche dieser Operatoren können **nur** mit Ganzzahlen verwendet werden?

- [ ] `+`
- [ ] `/`
- [ ] `-`
- [ ] `%`
- [ ] `*`

Verzweigungen

`if`-Anweisungen

Was gibt dieses Programm aus?

```
#include <iostream>

int main() {
  int a = 44;
  int b = 3;

  if (a == 44 && a == b) {
    std::cout << "1234\n";
  }
  else {
    if (a >= b || a == 10) {
      std::cout << "5678\n";
    }
    else {
      std::cout << "9\n";
    }
  }
  return 0;
}
```

`switch`-Anweisungen

Welche Zahlen dürfen zwischen den runden Klammern nach dem Schlüsselwort `switch` stehen?

- ◯ Ganzzahlen
- ◯ Gleitkommazahlen

Was gibt dieses Programm aus?

```
#include <iostream>

int main() {
  int b = 6;
  int a = b;

  switch(a) {
    case 4:
      std::cout << "4\n";
      break;
    case 5:
    case 6:
    case 7:
      std::cout << "5 bis 7\n";
    case 3:
      std::cout << "3\n";
      break;
    case 0:
      std::cout << "0\n";
    default: std::cout << "Keine Kategorie!\n";
  }
  return 0;
}
```

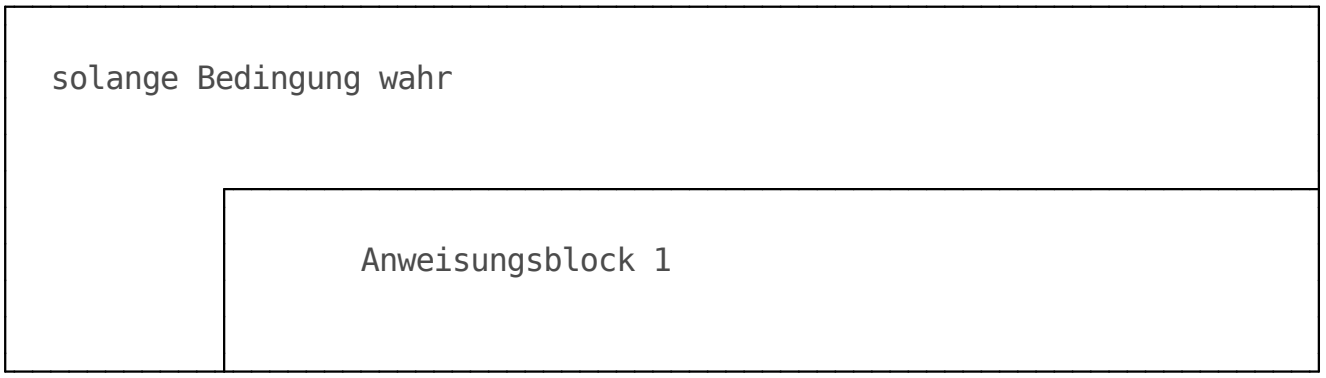
Was gibt dieses Programm aus?

```
#include <iostream>

int main() {
  int b = 9;
  int a = b;
```

```
  switch(a) {
    case 4:
      std::cout << "4\n";
      break;
    case 5:
    case 6:
    case 7:
      std::cout << "5 bis 7\n";
    case 3:
      std::cout << "3\n";
      break;
    case 0:
      std::cout << "0\n";
    default: std::cout << "Keine Kategorie!\n";
  }
  return 0;
}
```

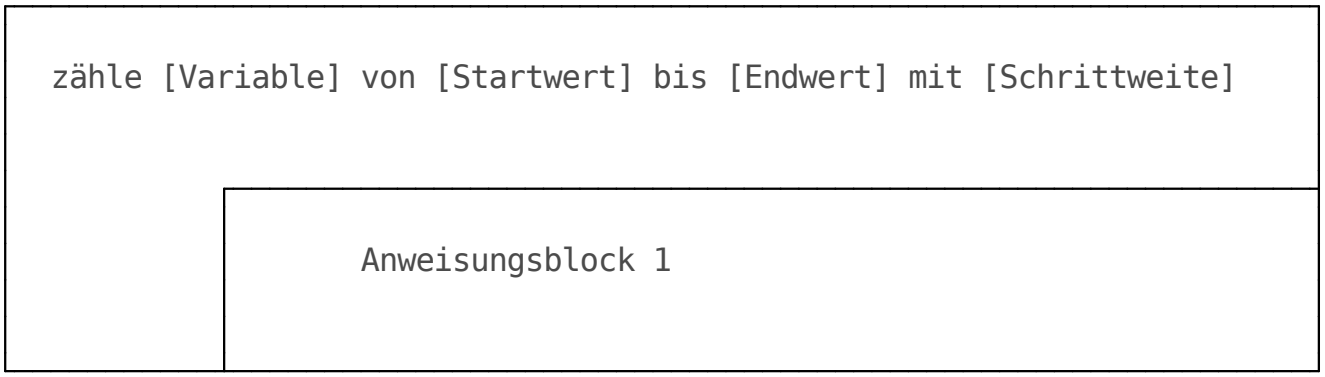
Schleifen

Welche Art von Schleife ist hier dargestellt?



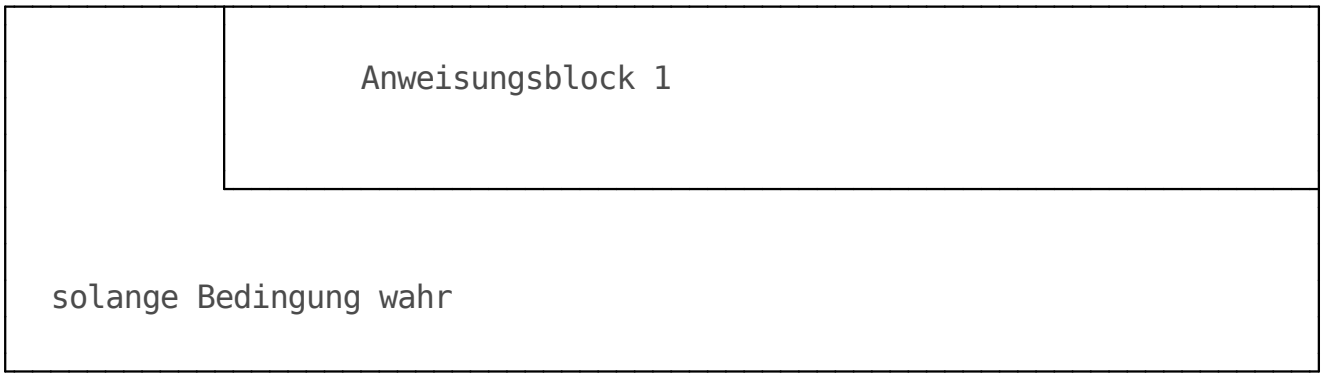
- ◯ `for`-Schleife
- ◯ `while`-Schleife
- ◯ `do-while`-Schleife

Welche Art von Schleife ist hier dargestellt?



- `for`-Schleife
- `while`-Schleife
- `do-while`-Schleife

Welche Art von Schleife ist hier dargestellt?



- `for`-Schleife
- `while`-Schleife
- `do-while`-Schleife

`for`-Schleife

Dieses Programm soll die Zahlen 4 bis 15 einzeln in aufsteigender Reihenfolge ausgeben.
Beantworten Sie die unten aufgeführten Fragen.

```
#include <iostream>

int main() {
```

```
int main() {
  int i;
  for (i = [_____]; i < [_____]; i++)
    std::cout << i << "\n";

  return 0;
}
```

Mit welchem Wert wird `i` initialisiert?

Welcher Wert muss in der Abbruchbedingung der Schleife stehen?

Welchen Wert hat `i` nach der Schleife?

`while`-Schleife

Wie lautet die Ausgabe dieses Programms?

```
#include <iostream>

int main() {
  int i = 16;
  while (i > 4) {
    i = i / 2;
    std::cout << i << " ";
  }

  std::cout << "ende\n";
  return 0;
}
```

Welcher Wert wird für die Variable `zaehler` ausgegeben wenn folgende Eingaben einzeln getätigt werden? `X` `X` `A` `X` `Y` `X` `Y`

```
#include <iostream>

int main() {
  char c;
  int zaehler = 0;
  std::cin >> c;
  while(c != 'Y')
  {
    if(c == 'X')
      zaehler++;
    std::cin >> c;
  }
  std::cout << zaehler << "\n";
  return 0;
}
```

`do-while`-Schleife

Wie lautet die Ausgabe dieses Programms?

```
#include <iostream>

int main() {
  int i = 16;
  do {
    i = i / 2;
    std::cout << i << " ";
  } while (i < 4);

  std::cout << "ende\n";
  return 0;
}
```

Kontrolliertes Verlassen von Anweisungen

Wie lautet die Ausgabe dieses Programms?

```
#include <iostream>

int main() {
  int i;
  for (i = 1; i<10; i++) {
    if (i > 5) break;
    std::cout << i << " ";
  }
  std::cout << "ende\n";
  return 0;
}
```

Wie lautet die Ausgabe dieses Programms?

breakForLoop.cpp

```
#include <iostream>

int main() {
  int i;
  for (i = 1; i<10; i++) {
    if (i < 5) continue;
    std::cout << i << " ";
  }
  std::cout << "ende\n";
  return 0;
}
```