Rechtlicher Hinweis

Legal notice

Informatik I (B.Sc. Physik)

Steueranweisungen / Kontrollstrukturen (a.k.a. Alternativen und Schleifen)

Dr. Paul Bodesheim
(Paul.Bodesheim@uni-jena.de)

FRIEDRICH-SCHILLER-
UNIVERSITÄT
JENA

Fakultät für Mathematik und Informatik
Lehrstuhl für Digitale Bildverarbeitung

SoSe 2020

Inhalt

Inhalt

Kontrollstrukturen

- Ganz wenige Programme bestehen nur aus einer linearen Folge von Anweisungen (feste Reihenfolge)
- Oft hängt der Ablauf von den Werten der verwendeten Variablen (Daten) ab

  $\Rightarrow$ Bedingungen steuern die Abarbeitung von Anweisungen
- Beispiel: unterschiedliche Verarbeitung, wenn eine Zahl positiv oder negativ ist
- Kontrollstrukturen verändern die lineare Abarbeitungsreihenfolge

Inhalt

Einfache Alternative (if)

- Verzweigung im Programmablauf durch **if-Anweisung** (*if-statement*)
- Ausschlaggebend ist Wahrheitswert einer Bedingung (logischer Ausdruck)

  Wenn "Bedingung erfüllt", dann ... , sonst ...

  if "condition fulfilled" , then ... , else ...

- Definition in C++:
  if (<BEDINGUNG>) <ANWEISUNG>; [**else** <ANWEISUNG>;]
- Schlüsselwörter: **if** , **else** (kein then!)

- **else** ist optional!

  $\Rightarrow$ Ohne **else**: Ausführen von zusätzlichen Anweisungen, die nur im Fall "Bedingung erfüllt" sinnvoll sind
- Beispiel: **if** (a<0) a = −a; // Absolutbetrag einer Zahl a

Besonderheiten

- Die Bedingung ist **immer** in Klammern zu setzen! (notwendige Konsequenz aus dem Fehlen von then)
- Nach der Bedingung genau eine Anweisung, die ausgeführt wird, wenn die Bedingung erfüllt ist
- Analog nach dem optionalen **else** genau eine Anweisung, die ausgeführt wird, wenn die Bedingung nicht erfüllt wird
- Bei mehreren Anweisungen in einer Verzweigung: Anweisungen zu Block zusammenfassen mit { ... }

Beispiele

Einzelne Anweisungen in gleicher Zeile

```
if (a<0) cout << "a ist negativ" << endl;
else cout << "a ist nicht negativ" <<endl;
```

Einzelne Anweisungen eingerückt in eigener Zeile

```
if (a<0)
    cout << "a ist negativ" << endl;
else
    cout << "a ist nicht negativ" <<endl;
```

Einzelne Anweisungen eingerückt in eigenem Block

```
if (a<0)
{
    cout << "a ist negativ" << endl;
}
else
{
    cout << "a ist nicht negativ" <<endl;
}
```

Erläuterungen

- Klammerung nicht notwendig, aber sinnvoll zur verbesserten Lesbarkeit
- **Eine gute Formatierung erleichtert das Lesen und Verstehen des Programms!**

- Vor allem wichtig bei verschachtelten Verzweigungen:

```
if (a>0)
{
    cout << "positiv" << endl;
}
else
    if (a<0)
    {
        cout << "negativ" << endl;
    }
    else
    {
        cout << "gleich 0" << endl;
    }
```

$\Rightarrow$ if-Anweisung ist syntaktisch eine einzelne Anweisung!

Beispiel: Dreiecksungleichungen

```
// double a,b,c; Eingaben mit cin
if ( a+b <= c )
    cout << "Kein Dreieck" << endl;
else
    if (b+c <= a)
        cout << "Kein Dreieck" << endl;
    else
        if (a+c <= b)
            cout << "Kein Dreieck" << endl;
        else
        {
            double umfang = a+b+c;
            double s = umfang/2.0;
            double flaeche = sqrt(s * (s-a) * (s-b) * (s-c));
        }
```

Beispiel: Dreiecksungleichungen (übersichtlicher)

- Gleiche Anweisungen für mehrere Bedingungen zusammenfassen
- Verknüpfen der Bedingungen mit logischen Operatoren

```
// double a,b,c; Eingaben mit cin
if ( a+b <= c || b+c <= a || a+c <= b)

    cout << "Kein Dreieck" << endl;

else
{
    double umfang = a+b+c;
    double s = umfang/2.0;
    double flaeche = sqrt(s * (s-a) * (s-b) * (s-c));
}
```

Beispiel: unterschiedliche Klammerung ("else if")

<ANWEISUNG-3> wird nur ausgeführt, wenn <BEDINGUNG-1> zu false ausgewertet wird

```
if (<BEDINGUNG-1>)
    <ANWEISUNG-1>;
else
{
    if (<BEDINGUNG-2>)
    {
        <ANWEISUNG-2>;
    }
    <ANWEISUNG-3>;
}
```

Mit "else if": <ANWEISUNG-3> wird immer ausgeführt

```
if (<BEDINGUNG-1>)
    <ANWEISUNG-1>;
else if (<BEDINGUNG-2>)
{
    <ANWEISUNG-2>;
}
<ANWEISUNG-3>;
```

Beispiel: unterschiedliche Klammerung ("else if") 2

<ANWEISUNG-3> wird nur ausgeführt, wenn <BEDINGUNG-1> zu false ausgewertet wird

```
if (<BEDINGUNG-1>)
    <ANWEISUNG-1>;
else
{
    if (<BEDINGUNG-2>)
    {
        <ANWEISUNG-2>;
    }
    <ANWEISUNG-3>;
}
```

Mit "else if": <ANWEISUNG-3> wird immer ausgeführt

```
if (<BEDINGUNG-1>)
    <ANWEISUNG-1>;
else
    if (<BEDINGUNG-2>)
    {
        <ANWEISUNG-2>;
    }
<ANWEISUNG-3>;
```

Beispiel: Klammerung vergessen

- Einrückung spielt für Gruppierung **keine** Rolle
- Einrückung verändert nicht den Syntax
- Einrückung ändert Programmablauf nicht
- Einrückung nur als Mittel zur Verbesserung der Lesbarkeit, Programmierer muss auf "Korrektheit" achten!

Was schnell passiert, wenn man die Klammern vergisst

```
if (<BEDINGUNG-1>)
    <ANWEISUNG-1>;
else if (<BEDINGUNG-2>)
        <ANWEISUNG-2>;
        <ANWEISUNG-3>;
```

Wenn die Einrückung die Gruppierung widerspiegelt

```
if (<BEDINGUNG-1>)
    <ANWEISUNG-1>;
else
    if (<BEDINGUNG-2>)
        <ANWEISUNG-2>;
<ANWEISUNG-3>;
```

$\Rightarrow$ <ANWEISUNG-3> wird immer ausgeführt

Inhalt

Schleifen

- Schleifen erlauben das wiederholte Ausführen von bestimmten Anweisungen! $\Rightarrow$ *Wiederholungsschleifen*
- Mehrfaches Ausführen der Anweisungen $\Rightarrow$ mehrere Durchläufe

- *Schleifenorganisation* durch 3 Komponenten:
  - **Initialisierung** (typischerweise einer "Schleifenvariable")
  - Testen einer **Schleifenbedingung** (Abbruchbedingung)
  - **Iteration** (Modifikation der "Schleifenvariable")

- *Aufbau* einer Schleife durch 2 Teile
  - **Schleifenkopf**: in der Regel zur Schleifenorganisation, je nach Schleifentyp aber nicht vollständig
  - **Schleifenrumpf**: enthält die Anweisungen, die wiederholt auszuführen sind

for-Schleife

- Schlüsselwort: **for**
- Definition:
  for (<INITIALISIERUNG>; <BEDINGUNG>; <ITERATION>)
  <ANWEISUNG>
- Schleifenorganisation in runden Klammern, besteht in der Regel aus 3 Teilen, die durch Semikolon getrennt werden
- <INITIALISIERUNG>: Initialisierung der Schleifenvariable
- <BEDINGUNG>: logischer Ausdruck, wird *vor* jedem Durchlauf überprüft

  $\Rightarrow$ Ausführen der <ANWEISUNG> nur, wenn <BEDINGUNG> erfüllt ist bzw. Wiederholtes Ausführen der <ANWEISUNG> solange <BEDINGUNG> erfüllt ist
- <ITERATION>: Verändern der Schleifenvariable *nach* jedem Durchlauf, z.B. Inkrement oder Dekrement
- <ANWEISUNG>: entweder einzelne Anweisung oder Block {...}

Beispiel: Addieren der Zahlen von 1 bis 100

```
int i;
int sum = 0;
for (i=1; i <= 100; i++)
    sum = sum + i;
```

Effektiver: Summenformel (Gauss)

Umgekehrte Reihenfolge:

```
int i;
int sum = 0;
for (i=100; i >= 1; i--)
    sum = sum + i;
```

Bei bestimmten Anwendungen kann die Reihenfolge (Richtung) eine Rolle spielen!

Beispiel: keine fortlaufenden Werte

```
int p;
int n = 100;
for (p=1; p < n; p=p+p) ;
```

- Was wird berechnet?
- Schleifenrumpf besteht aus “leerer Anweisung” (Semikolon)
- p enthält nach der Schleife die kleinste Zweierpotenz, die größer als eine gegebene Zahl n (hier: 100) ist

  $\Rightarrow$ Flexible Gestaltung der Iteration!

  $\Rightarrow$ “Schleifenvariable” zählt nicht die Anzahl der Wiederholungen

  $\Rightarrow$ Anzahl der Wiederholungen nicht unmittelbar ersichtlich

Lokale Schleifenvariable (Laufvariable)

```
int sum = 0;
for (int i=1; i <= 100; ++i)
    sum +=i;
```

- Die Variable i ist nur in der Schleife gültig!

  ⇒ Oftmals gewünscht, z.B. als Index für die Wiederholung oder für "Zwischenwerte", die später nicht mehr gebraucht werden

  ⇒ Laufvariablen haben oft nur lokale Bedeutung

  ⇒ Laufvariablen werden typischerweise im Schleifenrumpf nicht verändert

- Iteration: ++i oder i++ oder i=i+1 (kein Semikolon!) im Schleifenkopf
- Ergebnis wird nicht in Schleifenvariable gespeichert, sondern in sum (im Gegensatz zum vorherigen Beispiel mit p)
- Abkürzende Schreibweise: sum +=i; für sum = sum + i;

Mehrere Laufvariablen

- Addiere die Elemente einer 5×5-Matrix!

```
double sum = 0.0;
for (int row = 1; row <= 5; row++)
    for (int col=1; col <=5; col++)
        // Addiere Matrixelement in Zeile row und Spalte col
```

⇒ Berechnet Summe aller Matrixelemente

```
double sum = 0.0;
for (int row = 1, col=1; row <= 5 && col<=5; row++,col++)
    // Addiere Matrixelement in Zeile row und Spalte col
```

⇒ Berechnet Summe der Hauptdiagonalelemente

- Initialisierung mehrerer Schleifenvariablen (des gleichen Typs!) möglich, durch Komma getrennt
- Zusammengesetzte Bedingungen möglich, ein logischer Ausdruck
- Iteration kann mehrere Anweisungen umfassen (nicht nur von Schleifenvariablen), durch Komma getrennt

Leere Initialisierung und leere Iteration

Zahlen 1 bis 5 ausgeben

```
for ( int i=1; i <= 5; i++)
    cout << i << endl;
```

Leere Initialisierung

```
int i=1; // Initialisierung
for ( ; i <= 5; i++)
    cout << i << endl;
```

Leere Iteration

```
for ( int i=1; i <= 5; )
{
    cout << i << endl;
    i++; // Iteration
}
```

Leere Initialisierung und leere Iteration

```
int i=1; // Initialisierung
for ( ; i <= 5 ; )
{
    cout << i << endl;
    i++; // Iteration
}
```

- Alle 4 Varianten tun das gleiche!
- Was gilt es zu beachten?

Boolesche Variable als Bedingung

```
bool lowerThan5 = true;
for ( int i=1; lowerThan5 ; i++ )
{
    cout << i << endl;
    if (i==5)
        lowerThan5 = false;
}
```

- Bedingung muss logischer Ausdruck bzw. Wahrheitswert sein
  ⇒ Kann durch Variable repräsentiert werden
- Auch zusammengesetzte Ausdrücke mit booleschen Variablen möglich

```
int sum = 0;
bool isValid = true;
for ( int i=1; sum < 100 && isValid; i++ )
{
    sum += i;
    if (sum % 12 == 0)
        isValid = false;
}
```

Achtung: Gefahr einer Endlosschleife

- **Endlosschleife**: Programm wiederholt Schleifenrumpf unendlich oft
  $\Rightarrow$ Programm endet nie, führt nicht alle Anweisungen / Berechnungen bis zum Ende aus
- Verschiedene Ursachen möglich:

```
// Iteration in die falsche Richtung
double sum = 0.0;
for ( double d=10.0; d >= 0; d++ )
    cout << (sum += d) << endl;

// Iteration und Schleifenbedingung nicht abgestimmt
double sum = 0.0;
for ( double d=1.0; d >= 0; d/=2.0 )
    cout << (sum += d) << endl;

// Schleifenbedingung schlecht gewaehlt (immer wahr)
double sum = 0.0;
for ( double d=1.0; sum <= 2.0; d/=2.0 )
    cout << (sum += d) << endl;

// Variable in der Schleifenbedingung wird nicht veraendert
double sum = 0.0;
for ( double d=1.0; sum <= 1.8; d/=2.0 )
    cout << (sum + d) << endl;
```

while-Schleife

- Schlüsselwort: **while**
- Definition: **while** (<BEDINGUNG>) <ANWEISUNG>
- Schleifenbedingung in runden Klammern
- Wiederhole <ANWEISUNG> solange <BEDINGUNG> erfüllt (wahr) ist
- <BEDINGUNG>: logischer Ausdruck
- <ANWEISUNG>: einzelne Anweisung oder Block von Anweisungen
- Wichtig: Anweisungen im Schleifenrumpf müssen Einfluss auf Bedingung nehmen! (sonst Gefahr der Endlosschleife)

```
int p = 1;
int n = 100;
while (p < n)
    p=p+p; // p*=2;
cout << p << endl; // 128
```

do-while-Schleife

- Schlüsselwörter: **do**,**while**
- Definition: **do** <ANWEISUNG> **while** (<BEDINGUNG>);
- Schleifenbedingung in runden Klammern, beachte Semikolon danach!
- Wiederhole <ANWEISUNG> solange <BEDINGUNG> erfüllt (wahr) ist
- <BEDINGUNG>: logischer Ausdruck
- <ANWEISUNG>: einzelne Anweisung oder Block von Anweisungen
- Wichtig: Anweisungen im Schleifenrumpf müssen Einfluss auf Bedingung nehmen! (sonst Gefahr der Endlosschleife)

```
int p = 1;
int n = 100;
do
    p=p+p; // p*=2;
while (p < n);
cout << p << endl; // 128
```

Unterschied: Semikolon

```
while (<Bedingung-1>) // hier kein Semikolon! (kritisch)
{
    <Anweisung-1>
     ...
    <Anweisung-4>
} // hier auch kein Semikolon! (unkritisch)
...
do
{
    <Anweisung-5>
     ...
    <Anweisung-9>
} while (<Bedingung-2>); // Semikolon muss sein, sonst Fehler
```

- Kritisch: **while** (<Bedingung>); ist Endlosschleife mit leerer Anweisung als Schleifenrumpf!
- Unkritisch: Semikolon am Ende eines Blocks hängt einfach leere Anweisung an (tut nichts)

Unterschied: abweisende vs. nicht abweisende Schleife

while-Schleife ist abweisende Schleife

Schleifenrumpf wird ggf. nicht ein einziges Mal ausgeführt:

```
int p = 101;
int n = 100;
while (p < n)
    p=p+p; // wird nicht ausgefuehrt
cout << p << endl; // 101
```

do-while-Schleife ist nicht-abweisende Schleife

Schleifenrumpf wird mindestens einmal ausgeführt:

```
int p = 101;
int n = 100;
do
    p=p+p; // wird ausgefuehrt
while (p < n);
cout << p << endl; // 202
```

Wie sieht es mit der for-Schleife aus?

for-Schleife ist auch abweisende Schleife

```
int p = 101;
int n = 100;
for (; p < n; )
    p=p+p; // wird nicht ausgefuehrt
cout << p << endl; // 101
```

Auch die Iteration im Schleifenkopf wird ggf. nicht ausgeführt

```
int p = 101;
int n = 100;
for (; p < n; p=p+p) ;
    // Iteration wird nicht ausgefuehrt
cout << p << endl; // 101
```

Unterbrechungen von Schleifen

- Schlüsselwörter: **break**,**continue**
- Beide werden genutzt, um aktuellen Ablauf (Abarbeitung der Anweisungen) im Schleifenrumpf zu unterbrechen
- Unterschiedliche Auswirkung/Arbeitsweise:

break

- Abbruch / Verlassen der Schleife
- Abarbeitung des Schleifenrumpfes endet unmittelbar bei **break**
- Bei verschachtelten Schleifen (z.B. zwei for-Schleifen): nur Abbruch und Verlassen der innersten Schleife

continue

- Der Rest des Schleifenrumpfes nach **continue** wird nicht abgearbeitet
- Schleife wird aber fortgesetzt (nächste Iteration) mit Testen der Schleifenbedingung

  $\Rightarrow$ bei **for**-Schleife wird zuvor auch Iterationsanweisung im Schleifenkopf ausgeführt

Beispiele: break und continue

```
for (int a = 0; a <= 5; a++)
{
    cout << a;
    if (a>2) break;
    cout << " + ";

}
```

⇒ Ausgabe: 0+1+2+3

```
for (int a = 0; a <= 5; a++)
{
    cout << a;
    if (a>2) continue;
    cout << " + ";

}
```

⇒ Ausgabe: 0+1+2+345

Beispiel: “gewollte Endlosschleife”

```
int i=-1;
while (true)
{
    cin >> i;

    // Tue etwas mit i
}
```

Beispiel: “gewollte Endlosschleife”

```
int i=-1;
while (true)
{
    cin >> i;
    if (i == 0) break;
    // Tue etwas mit i
}
```

- Zusätzliche Bedingungen schaffen, die zum Abbruch der Schleife führen können
- Alternative:

```
int i=-1;
while (i!=0)
{
    cin >> i;
    if (i!=0)
    {
        // Tue etwas mit i
    }
}
```

Beispiel: “ungewollte Endlosschleifen”

```
int i=0;
while (i<10)
{
    // Tue etwas mit i
    ...
    // Inkrement von i (i++) vergessen
}


double j=0;
while (j!=1.0)
{
    // Tue etwas mit j
    ...
    j+=0.1;
    cout << j << endl;
    // 1.0 wird nie exakt erreicht,
    // da 0.1 eine periodische Darstellung
    // im Binaersystem hat, bessser:
    // Schleifenkopf als while ( abs(j-1.0)>1e-10 )
    // oder im Schleifenrumpf:
    // if ( abs(j-1.0) < 1e-10 ) break;
}
```

Inhalt

Mehrfache Alternative (switch)

- Schlüsselwörter: **switch**, **case**, **default**, **break**
- Ausgangspunkt: Ausdruck von ordinalem Typ, Wert entscheidet über Verzweigung
- **switch**: definiert den Ausdruck, anhand dessen die Entscheidung gefällt wird
- **case**-Anweisungen: legen fest, welche Befehle bei welchem Wert ausgeführt werden (wichtig: Ausführung bis zum nächsten **break**)
- **case**-Anweisungen sind "Sprungmarken"
- Beispiel: verbale Einschätzung für Noten

```
int note;
cin >> note;
switch (note) // Ausdruck von Typ int
{
    case 1: cout << "sehr gut" << endl; break;
    case 2: cout << "gut" << endl; break;
    case 3: cout << "befriedigend" << endl; break;
    case 4: cout << "ausreichend" << endl; break;
    case 5: cout << "ungenuegend" << endl; break;
}
```

Abfangen ungültiger Eingaben (Noten)

- Problem: nicht alle Integer-Werte können als Fälle für **case**-Anweisungen aufgelistet werden
- Abfangen aller anderen Werte: **default**

```
int note;
cin >> note;
switch (note) // Ausdruck von Typ int
{
    case 1: cout << "sehr gut" << endl; break;
    case 2: cout << "gut" << endl; break;
    case 3: cout << "befriedigend" << endl; break;
    case 4: cout << "ausreichend" << endl; break;
    case 5: cout << "ungenuegend" << endl; break;
    default: cout << "ungueltige Note" << endl; break;
}
```

Verändertes Beispiel

- Eingabe: Note
- Ausgabe: bestanden / nicht bestanden

```
int note;
cin >> note;
switch (note) // Ausdruck von Typ int
{
    case 1: cout << "bestanden" << endl; break;
    case 2: cout << "bestanden" << endl; break;
    case 3: cout << "bestanden" << endl; break;
    case 4: cout << "bestanden" << endl; break;
    case 5: cout << "nicht bestanden" << endl; break;
    default: cout << "ungueltige Note" << endl; break;
}
```

- Geht das nicht einfacher?
- **case** erlaubt nicht mehrere Werte oder ganzen Wertebereich

Kürzere Schreibweise

```
int note;
cin >> note;
switch (note) // Ausdruck von Typ int
{
    case 1:
    case 2:
    case 3:
    case 4: cout << "bestanden" << endl; break;
    case 5: cout << "nicht bestanden" << endl; break;
    default: cout << "ungueltige Note" << endl; break;
}
```

- Zusammenfassen von gleichen Anweisungen durch fehlende **break**-Anweisungen
- Auch für Fälle 1-3 erfolgt Abarbeitung bis zum nächsten **break**

Gewolltes “Durchfallen”

```
int note;
cin >> note;
switch (note) // Ausdruck von Typ int
{
    case 1: cout << "sehr "; // no break
    case 2: cout << "gut" << endl; break;
    case 3: cout << "befriedigend" << endl; break;
    case 4: cout << "ausreichend" << endl; break;
    case 5: cout << "ungenuegend" << endl; break;
    default: cout << "ungueltige Note" << endl; break;
}
```

- “Durchfallen” zum zweiten case hier erwünscht
- Gefahr: ungewolltes “Durchfallen”, wenn **break** vergessen wird
- Konvention: fehlendes **break** explizit in Kommentar erwähnen

Weitere Anmerkungen

- Werte für den Ausdruck (**case**-Anweisungen) müssen nicht sortiert / geordnet sein
- Es müssen nicht alle Werte zwischen Minimum und Maximum abgedeckt werden
- **default**-Anweisung muss nicht am Ende kommen, ist aber üblich
- **switch** lässt sich auch nur mit **if**-Anweisungen umsetzen

- Was es noch gibt: **goto**-Anweisung mit Sprungmarken
- Aber: **goto** ist böse (kein guter Stil, unübersichtlich, fehleranfällig, Code ist schwerer nachzuvollziehen, ...)

https://www.xkcd.com/292/

Überführbarkeit

- Jedes Programm mit **goto** lässt sich auch durch Programm ohne **goto** mit Schleifen und Alternativen realisieren
- Jede Schleife (**for**,**while**,**do**−**while**) lässt sich auch semantisch äquivalent durch jede andere Schleife implementieren

  ⇒ Es besteht freie Wahl, welche Schleifen verwendet werden

```
int sum = 0;

for (int i=1; i<100; i++)
{

    sum +=i;


}

cout << sum;
```

```
int sum = 0;
int i = 1;

while (i<100)
{

    sum += i;
    i++;

}

cout << sum;
```

```
int sum = 0;
int i = 1;

if (i<100)
    do
    {
        sum += i;
        i++;

    } while (i<100);

cout << sum;
```

Überführbarkeit (2)

- Jedes Programm mit **goto** lässt sich auch durch Programm ohne **goto** mit Schleifen und Alternativen realisieren
- Jede Schleife (**for**,**while**,**do**−**while**) lässt sich auch semantisch äquivalent durch jede andere Schleife implementieren

  ⇒ Es besteht freie Wahl, welche Schleifen verwendet werden

```
int sum = 0;
int i = 1;

for ( ; i<100; )
{

    sum +=i;
    i++;

}

cout << sum;
```

```
int sum = 0;
int i = 1;

while (i<100)
{

    sum += i;
    i++;

}

cout << sum;
```

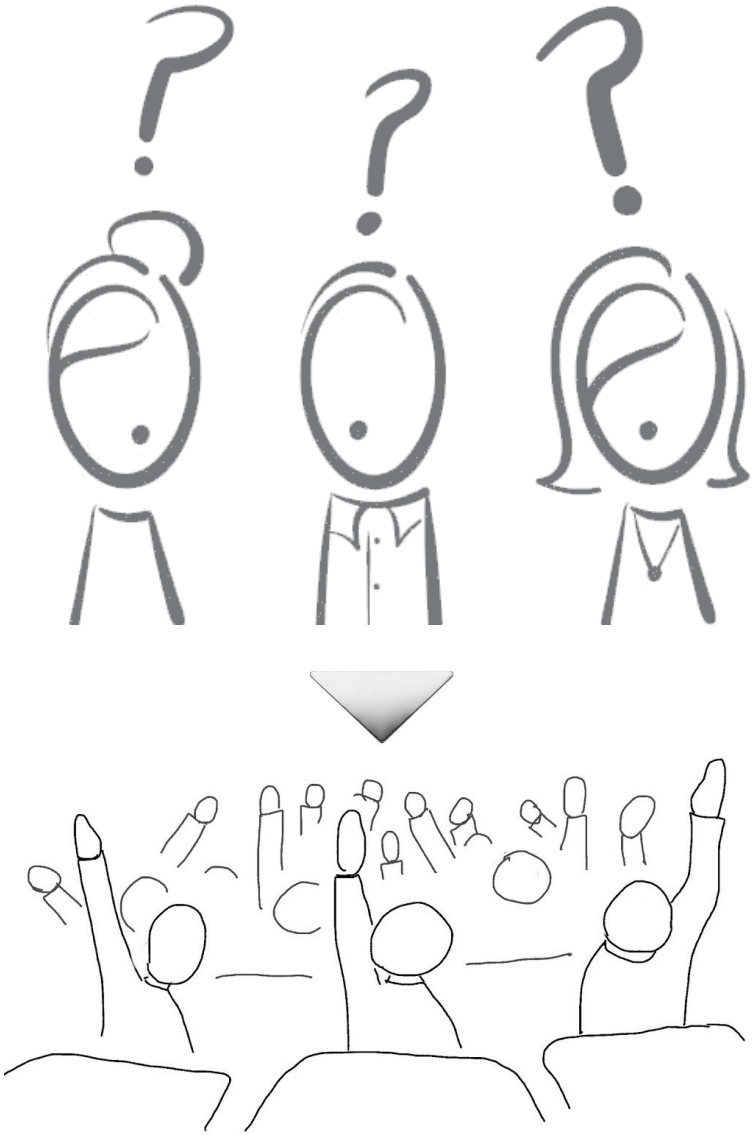
```
int sum = 0;
int i = 1;

if (i<100)
    do
    {
        sum += i;
        i++;

    } while (i<100);

cout << sum;
```

Gibt es Fragen? (Es gibt keine dummen Fragen!)



"Excuse me, is this the Society for Asking Stupid Questions?"