Karlsruher Institut für Technologie

Institut für Technik der Informationsverarbeitung

Institutsleitung
Prof. Dr.-Ing. K. D. Müller-Glaser
Prof. Dr.-Ing. J. Becker
Prof. Dr. rer. nat. W. Stork

Übung02: Informationstechnik (IT)

Tobias Schwalb & Michael Tansella

Institut für Technik der Informationsverarbeitung (ITIV)

Teil2: Zeiger, Arrays und Kontrollstrukturen

KIT – Universität des Landes Baden-Württemberg und
nationales Forschungszentrum in der Helmholtz-Gemeinschaft

www.kit.edu

Inhalt: Übung02 – Teil 2

Karlsruher Institut für Technologie

1. Besprechung Übungsaufgaben 1.01-1.05
2. Adressvariablen und Referenzen
3. Arrays
4. Verzweigungen
5. Schleifen

Adressvariablen: Deklaration

- Auch Zeiger oder engl. Pointer genannt

- Speichern die Adressen von Speicherzellen oder Registern
- Deklaration durch Anhängen von `*` an den Variablentyp

- Beispiel:
  - Normale Variable: `int wertvariable;`
  - Adressvariable: `int* adressvariable;`

- Es gibt zu jedem Variablentyp (auch für die Selbsterstellten) einen entsprechen Adressvariablentyp

- Belegt unabhängig vom Typ immer den gleichen Speicherplatz
  - Je nach PC und Betriebssystem (im Allgemeinen 4 Byte – 32 Bit)

Adressvariablen: Zuweisung

- Speichern von Adressen in Adressvariablen
  - Zuweisung von Adressen mit Hilfe von `&` Operator
  - Merksatz: `&` bedeutet „Adresse von“
- `&` vor Variable, ist die Speicheradresse der Variablen

- Beispiel:

```
int wertVariable = 1000;
int* adressVariable;
adressVariable = &wertVariable;
```

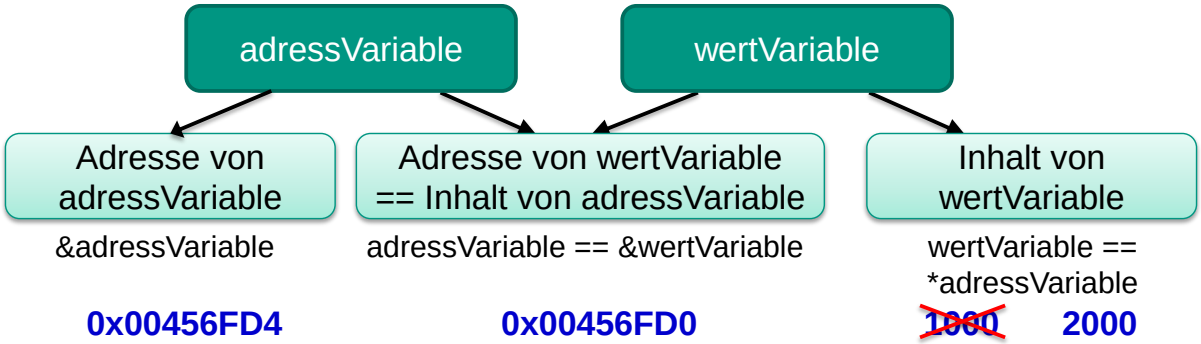
Variable	Inhalt der Variablen	Adresse (hex)
`wertVariable`	`1000`	`0x00456FD0`
`adressVariable`	`0x00456FD0`	`0x00456FD4`

Adressvariablen: Zugriff

- Um die Variable zu erhalten, auf welche ein Zeiger verweist, verwendet man den * Operator (Dereferenzierung)
- * vor Adressvariable, wird zum Inhalt, welcher an dieser Adresse gespeichert ist

- Beispiel:

```
int wertVariable = 1000;
int* adressVariable = &wertVariable;
*adressVariable = *adressVariable * 2;
```



Referenzen

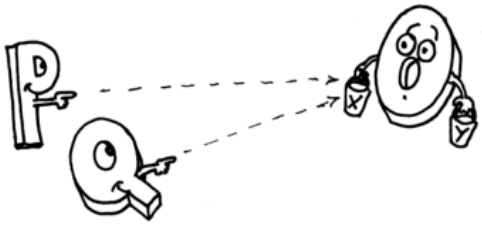
- Problemstellung:
  - Wie kann man auf eine Variable mit mehreren Namen zugreifen?

- Lösung: Referenzen
  - Kann direkt wie die Originalvariable verwendet werden
  - Bei Funktionen sehr nützlich (siehe Übung 3)

- Deklaration mit & „Ampersand“
- Beispiel:

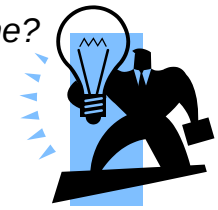
```
int wert = 10.7;
int& verweis = wert;
```

- Keine Variable → belegt keinen Speicher
  - Muss initialisiert werden
  - Später nicht mehr veränderbar



Zwischenübung04: Zeiger & Referenzen

Worin unterscheiden sich die Ausgaben der folgenden Programme?



```
int main() {
   int intOne;
   int* SomeAdr = &intOne;

   intOne = 5;
   cout << "intOne: "
        << intOne << endl;
   cout << "SomeAdr: "
        << SomeAdr << endl;
   cout << "Adr. von intOne: "
        << &intOne << endl;
   cout << "Adr. von SomeAdr: "
        << &SomeAdr << endl;
   return 0;
}
```

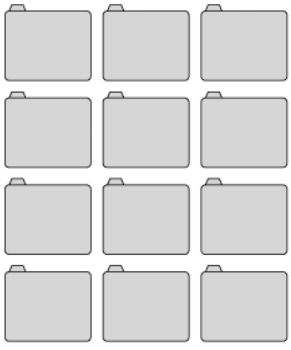
```
int main() {
   int intOne;
   int& SomeRef = intOne;

   intOne = 5;
   cout << "intOne: "
        << intOne << endl;
   cout << "SomeRef: "
        << SomeRef << endl;
   cout << "Adr. von intOne: "
        << &intOne << endl;
   cout << "Adr. von SomeRef: "
        << &SomeRef << endl;
   return 0;
}
```


Arrays

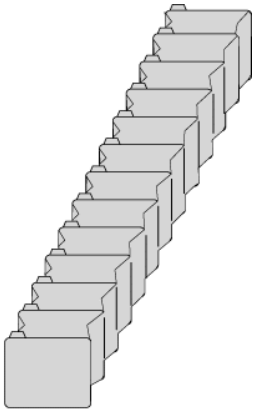
- Problemstellung:
  - Es sollen 100 Messwerte eines Experiments gespeichert werden
  - Anlegen von 100 Variablen des Typs **`float`** → viel Schreibarbeit ☹
- Lösung: Arrays
  - Anlegen einer Vielzahl von Variablen des **gleichen** Datentyps
  - Unterscheidung über den Index
- Syntax: **`typ name[anzahl];`**

Beispiel:

```
float arr[100];
```



Einzelne Variablen



Ein Array

Arrays: Zugriff

- Sind hintereinander im Speicher abgelegt
- Zugriff auf die einzelnen Arrayelemente über Indexoperator `[]`
  - Der Index beginnt immer mit 0, das letzte Element ist Länge-1
  - Für den Index können nur Ganzzahlen verwendet werden
  - Programmierer muss auf die Einhaltung der Grenzen achten
- Beispiel:

```
short index = 0;
int zahlarr[3] = {1,2,3};
zahlarr[index] = 1234;
index = 2;
zahlarr[index] = 554;
zahlarr[3] = 763;
```

zahlarr[0]	~~1~~ 1234
zahlarr[1]	2
zahlarr[2]	~~3~~ 554

Laufzeitfehler, da Element nicht vorhanden (wird evtl. erst später im Programm entdeckt)

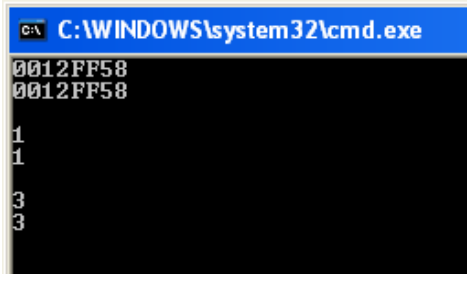
Arrays und Zeiger

- Arrayname = Zeiger auf das erste Arrayelement
- Beispiel:

```
int arr[3] = { 1, 2, 3 };
int* arrayZeiger = arr; // = &arr[0];
```

- `arr` und `arrayZeiger` sind Integer-Zeiger auf `arr[0]`, an dieser Stelle ist der Wert `1` gespeichert
  - `arr` ist eine Konstante → unveränderbar
- Beispiel:

```
cout << arrayZeiger << endl;
cout << arr << endl << endl;

cout << *arrayZeiger << endl;
cout << *arr << endl << endl;

cout << *(arrayZeiger + 2) << endl;
cout << *(arr + 2) << endl << endl;
```



Arrays und Zeiger – Rechnen mit Zeigern

- Zugriff auf das Array auch über Zeiger möglich (schreiben / lesen)
  - `arr[2]` ist gleichbedeutend mit `*( arr + 2 )`
  - Es werden auch hier keine Grenzen überprüft
  - Größe der Rechenschritte im Speicher wird entsprechend dem Typ des Zeigers automatisch angepasst

- Beispiel: `int arr[3] = { 1, 2, 3 };`

```
int* arrayZeiger = arr;
arrayZeiger = arrayZeiger + 2;
```

Variable	Inhalt der Variablen	Adresse (hex)
`arr[0]`	1	0012FF50
`arr[1]`	2	0012FF54
`arr[2]`	3	0012FF58
`arrayZeiger`	~~0012FF50~~ 0012FF58	0012FF40

Zeigerarithmetik und Read-Only Zeiger

- Arithmetische Operationen und Vergleiche sind mit Zeigern möglich
  - Operatoren `++, --, +=, -=, …`
  - Auf Operatorenreihenfolge achten (oder Klammern setzen)

- Subtraktion von Zeigern zur Bestimmung des Index in Arrays
  - Addition von Zeigern nicht zulässig, da auch nicht sinnvoll
- Beispiel:

```
int index = arrayzeiger – arr;
```

index = 2

- Ein Read-Only-Zeiger kann nur lesend auf Objekte zugreifen
  - Deklaration wie bei Variablen mit `const`
  - Zeiger selbst kann verändert werden (worauf er zeigt)
- Beispiel:

```
const int* cptrarr = arr;
```

Mehrdimensionale Arrays

- Problemstellung:
  - Wie kann ich eine 2-dimensionale Matrix speichern?
- Lösung:
  - Arrays können mehrere Dimensionen haben
- Beispiel:
  - Matrix hat 4 Zeilen und 9 Spalten
  - In Zeile 1 Spalte 3 steht der Wert 9.65

```
float zahl[4][9];
zahl[1][3] = 9.65;
```

- Initialisierung bei Definition möglich
- Beispiel:

```
float num[2][3] = {{ 30.2, 50.7, 60.9 },
                   { 12.2, 5.6, 99.0 }};
```

[↓][→]	0	1	2
0	30.2	50.7	60.9
1	12.2	5.6	99.0

Casting von Variablen

- Zuweisung des Wertes einer Variablen zu einer andern Variablen mit unterschiedlichem Typ
- Beispiel:

```
int ganzzahl = 10;
double* zeiger;
double fliesskomma;

fliesskomma = ganzzahl;

ganzzahl = fliesskomma;

ganzzahl = ( int ) fliesskomma;

zeiger = fliesskomma;
```

OK, Implizites Casting, da `double` eine höhere Genauigkeit hat als `int`

Ungültig, es geht Genauigkeit verloren

OK, explizites Casting – Genauigkeit geht verloren, aber dies ist dem Entwickler bewusst

Ungültig, `double*` und `double` sind nicht kompatibel

- Beim expliziten Casting wird der Zielvariablentyp in Klammern vor die umzuwandelnde Variable geschrieben
  - Voraussetzung ist, dass definiert ist, wie der Typ umgewandelt wird

Zwischenübung05: Arrays

Definieren Sie ein Array, das …

- das Monatsgehalt von 20 Angestellten speichern kann. Die ersten beiden Angestellten verdienen 3000.0 Euro.
- fünf Ganzzahlen speichern kann. Als Anfangswert erhält jedes Element das Doppelte seines Indexwertes.
- eine Ausgangsspannung in Abhängigkeit von 2 Widerstands-werten, 2 Eingangsspannungswerten und 3 Ausgangsstromwerten speichern kann.

Kontrollstrukturen

- Problemstellung:
  - Programm soll in Abhängigkeit von Bedingungen Entscheidungen treffen und dementsprechend Anweisungen ausführen
  - Anweisungen sollen oft mehrfach ausgeführt werden, wobei die Anzahl abhängig von einer Bedingung sein kann
- Beispiel:

  Falls ich heute Abend noch Zeit habe
  dann bearbeite ich noch die Übungsaufgaben
  andernfalls werde ich die Übung morgen lösen

  Solange der eingegebene Wert ungültig ist
  Frage nach einem neuen Wert

Bedingungen

- Frage stellen
  - Frage muss überprüfbar sein
  - Antwort darf nur ja (`true`) oder nein (`false`) sein

- Formulieren mit Vergleichsoperatoren
- Bedingungen verketten mit logischen Operatoren
- Ergebnisse können in `bool` - Variablen gespeichert werden
  - Es gilt: `0` ←→ `false`, `alles andere` → `true`, `true` → 1

- Beispiel: „Ist a mindestens so groß wie b ?“ lautet in C++: `a >= b`

```
int a = 5, b = 3;
bool vergleichsErgebnis = 3;
vergleichsErgebnis = (a >= b) && ( a == 4 ) || ( a < true );
```

== true

== false

Zuweisung

Vergleich

Vergleichsoperatoren

Operator	Bedeutung
<	kleiner
<=	kleiner gleich
>	größer
>=	größer gleich
==	gleich
!=	ungleich

Beispiel:

Vergleich	Ergebnis
`5 >= 6`	false
`1.7 < 1.8`	true
`( 4 + 2 ) == 5`	false
`( 2 * 4 ) != 7`	true

Logische Operatoren

Operator	Bedeutung
`&&`	UND
`||`	ODER
`!`	NICHT

Wahrheitstafel:

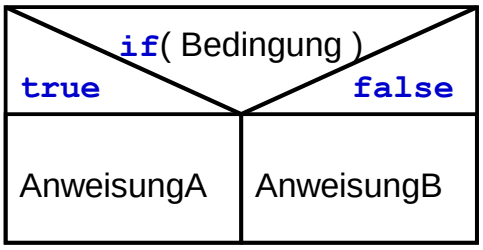
A	B	!A	A && B	A || B
`true`	`true`	`false`	`true`	`true`
`true`	`false`	`false`	`false`	`true`
`false`	`true`	`true`	`false`	`true`
`false`	`false`	`true`	`false`	`false`

Verzweigungen

- Festlegen, welche Anweisung als nächstes ausgeführt wird
- Syntax:

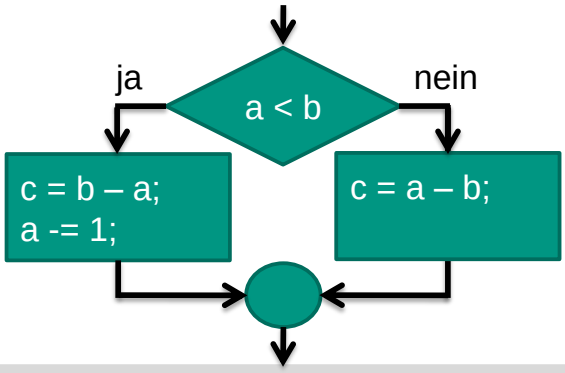
```
if( Bedingung )
   AnweisungA
[ else
   AnweisungB ]
```

Optional



- Mehrere Anweisungen müssen von `{ }` eingeschlossen werden
- Beispiel:

```
if( a < b ) {
   c = b - a;
   a -= 1;
}
else {
   c = a - b;
}
```

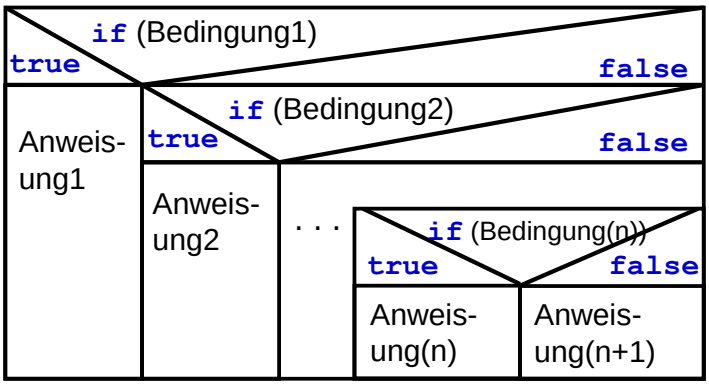


Mehrfachverzweigungen

- `if`-Verzweigungen können auch geschaltet werden
- Abfragen mit Prioritäten können abgebildet werden

- Syntax:

```
if( Bedingung1 )
  Anweisung1
else if( Bedingung2 )
  Anweisung2
      .
      .
else if( Bedingung( n ) )
  Anweisung( n )
[ else
  Anweisung( n+1 ) ]
```

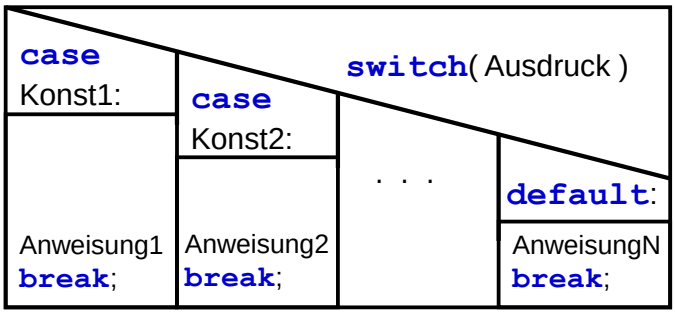


Mehrfachentscheidung

- Vergleich einer Ganzzahl-Variablen gegen eine Reihe fester Werte
  - Sprung an die Stelle im Programmcode entsprechend der Konstanten

- Syntax

```
switch( Ausdruck ) {
  case Konst1: [Anweisung1]
               [break; ]
  case Konst2: [Anweisung2]
               [break; ]
  . . .
  [default: AnweisungN ]
}
```



- **`Ausdruck`** → nur Ganzzahlvariablen (oder **`char`**) möglich
- **`Konst`** → nur Konstanten (Ganzzahlen, Zeichen) - keine Verkettung

- Endet bei den Anweisungen erst mit einem expliziten **`break;`**

Mehrfach… - Vergleich (Beispiel)

```
if( machineSt == 1 ) {
  continueWorking();
} else {
  if( machineSt == 0 ) {
    startWorking();
    calibrate();
  } else {
    if( machineSt == 2 ) {
      calibrate();
    } else {
      if( machineSt == -1 ) {
        stopWorking();
      }
      callService();
    }
  }
}
```

Diese beiden Programmabschnitte erfüllen die gleiche Aufgabe!

```
switch( machineSt ) {
  case 1:
    continueWorking();
    break;
  case 0:
    startWorking();
  case 2:
    calibrate();
    break;
  case -1:
    stopWorking();
  default :
    callService();
}
```


Zwischenübung06: Verzweigungen

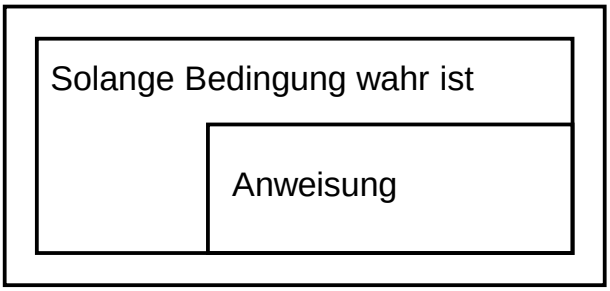
Berechnen Sie den Betrag der Differenz von a und b: |a-b| mit Hilfe einer Verzweigung und ergänzen Sie das folgende Programm entsprechend.

```
#include <iostream>
using namespace std;

int main() {
  int betrag, a, b;
  cout << "a und b eingeben:";
  cin >> a >> b;

        //Hier ergänzen




  cout << " |a-b| = " << betrag << endl;
  return 0;
}
```

While - Schleife

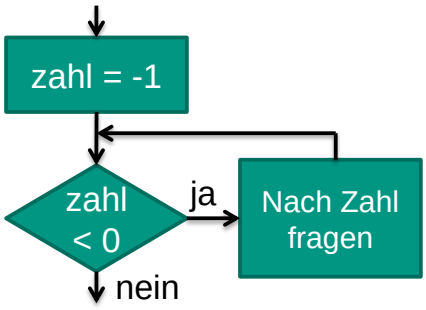
- Viele Sachen müssen mehrfach wiederholt werden
  - Feste Anzahl Wiederholungen oder abhängig von einer Bedingung
- **`while`**-Schleife - Kopfgesteuert
  - Bedingung **`true`** oder **`false`**
- Syntax:

```
while( Bedingung ) {
    Anweisung
}
```



- Beispiel:

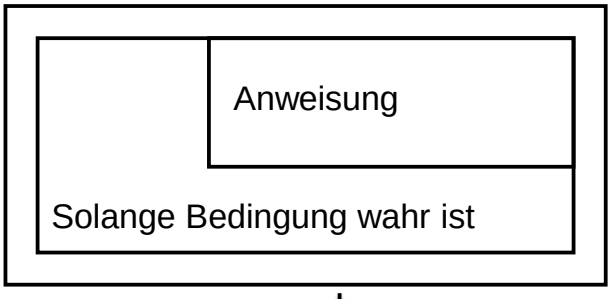
```
int zahl = -1;
while( zahl < 0 ) {
  cout << "Bitte geben Sie eine";
  cout << "Zahl größer Null ein: ";
  cin >> zahl;
}
```



Do-While - Schleife

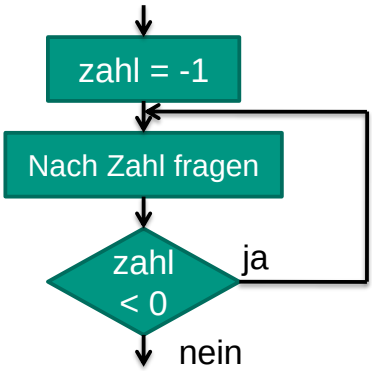
- Auch Fußgesteuerte Schleife genannt
  - Wird mindestens einmal aufgeführt (Unterschied zur **`while`**-Schleife)
  - Mehrere Anweisungen mit **`{ }`**
- Syntax:

```
do
    Anweisung
while( Bedingung );
```



- Beispiel:

```
int zahl = -1;
do {
  cout << "Bitte geben Sie eine";
  cout << "Zahl größer Null ein: ";
  cin >> zahl;
} while( zahl < 0 );
```

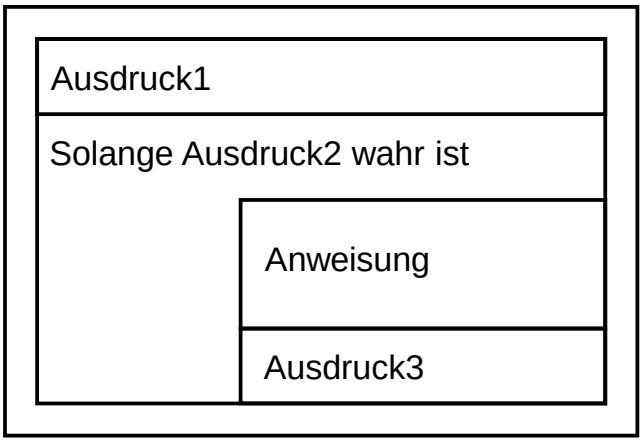


For – Schleife(1)

- Oft verwendet bei fester Anzahl an Durchläufen oder wenn eine Zählvariable benötigt wird
- Syntax:

```
for( Ausdruck1; Ausdruck2; Ausdruck3 ) {
        Anweisung
}
```

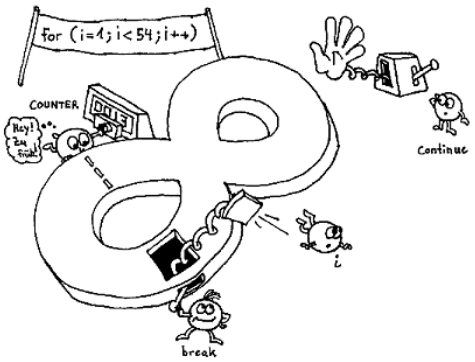
- Ausdruck1 = Initialisierung
  - Beginn der Schleife einmalig ausgeführt
- Ausdruck2 = Bedingung
  - Bestimmt wie lange die Schleife läuft
- Ausdruck3 = Veränderung
  - Am Ende jedes Schleifen-Durchlaufs ausgeführt



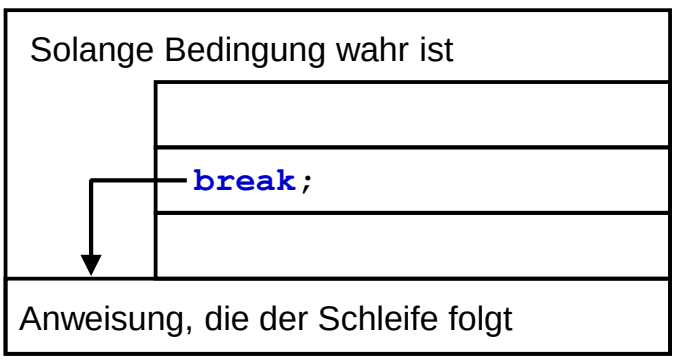
For – Schleife(2) und break;

- Beispiel:

```
for( int i = 1; i < 54; i++ ) {
    cout << i << " " << i * i;
    cout << endl;
    if( i * i > 1000 ) break;
}
```



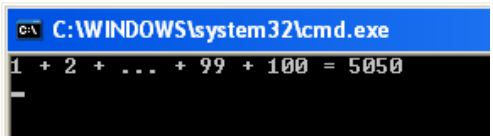
- Um eine Schleife sofort zu verlassen kann der Befehl `break;` verwendet werden
  - Nützlich für besondere Ereignisse und Error-Behandlung



Zwischenübung07: Schleifen

Ergänzen Sie das folgende Programm um eine Schleife, um die Summe der ersten 100 positiven Zahlen zu berechnen.

```
#include <iostream>
using namespace std;

int main() {
  unsigned long sum = 0;

  //Hier ergänzen


  cout << "1 + 2 + ... + 99 + 100 = " << sum << endl;

  return 0;
}
```



Referenz & Ausblick

- Kompendium: Kapitel 2 - Variablen, Zeiger und Arrays (ab 2.9)
  Kapitel 3 – Kontrollstrukturen (komplett)
- Tutorium: Aufgabe 5 - 8

- Wie kann ich ein Programm in mehrere Teile gliedern?
- Wie kann ich den gleichen Programmablauf an verschiedenen Stellen mit unterschiedlichen Daten wiederholen?

- Wie lange kann ich auf Variablen zugreifen, wo sind Sie sichtbar?

- …

Vielen Dank für Ihre Aufmerksamkeit

Tobias Schwalb & Michael Tansella

Karlsruher Institut für Technology (KIT) – ITIV
tobias.schwalb@kit.edu
michael.tansella@kit.edu