

Übung01: Informationstechnik (IT)

Harald Bucher

Institutsleitung

Prof. Dr.-Ing. Dr. h.c. J. Becker

Prof. Dr.-Ing. E. Sax

Prof. Dr. rer. nat. W. Stork

Institut für Technik der Informationsverarbeitung (ITIV)



Variablen & Gültigkeitsbereich, Operatoren, Kontrollstrukturen, Arrays

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

www.kit.edu

Inhalt: Übung01 Teil2

1

- Variablen

2

- Operatoren

3

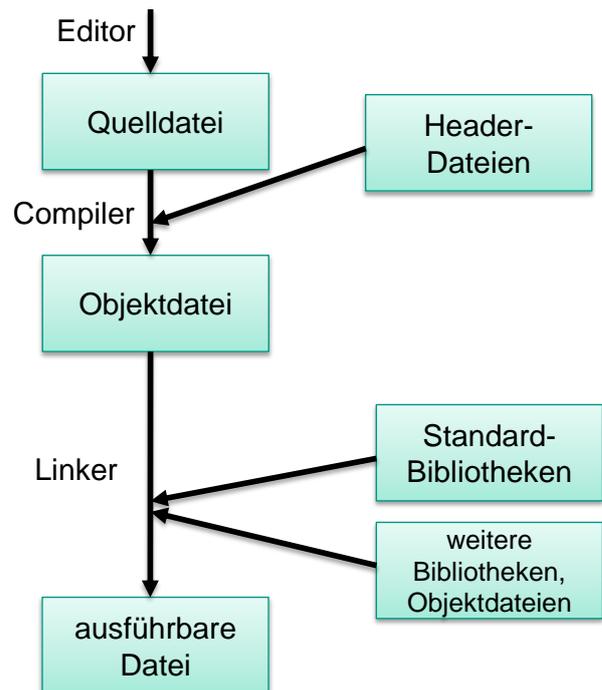
- Arrays

4

- Kontrollstrukturen

Erstellen eines C++ Programms

- Editor dient zur Eingabe des Programmcodes
- Mehrere Quelldateien & Headerdateien sind möglich
- Objektdatei enthält den Maschinencode
- Linker führt alle Bausteine zur einer ausführbaren Datei zusammen
- Übliche Dateiendungen:
 - *.cpp >> Quelldatei
 - *.h >> Headerdatei
 - *.obj >> Objektdatei
 - *.exe >> ausführbare Datei



Erstes C++ Programm

```
// Mein erstes C++ Programm
```

Kommentar bis zum Zeilenende mit //

```
#include <iostream>
```

kopiert die Datei <iostream> an diese Stelle
 → C++ Header-Datei für Ein- und Ausgabe

```
using namespace std;
```

Definiert den vorgegebenen Namensbereich
 Standard zu benutzen

```
int main()
```

Ein C++ Programm beginnt immer mit der ersten
 Anweisung in der Funktion `main()`

```
{
```

Mehrzeilige Kommentare zwischen /* und */

```
/* Die ganze Funktionalität dieses Programms */
```

```
cout << "Hello World" << endl;
```

```
return 0;
```

Befehle werden mit Semikolon abgeschlossen

```
}
```

Ausgabe auf die Konsole

```
cout << "Hello World" << endl;
```

Ausgabefunktion

- `cout` = console output
- `<<` = Zeichen in den Ausgabestrom schieben
- `endl` = end of line
- Text steht in doppelten Anführungszeichen



Achtung: Ausgabe bestimmter Zeichen nur über Escape-Sequenzen (siehe Kompendium) möglich

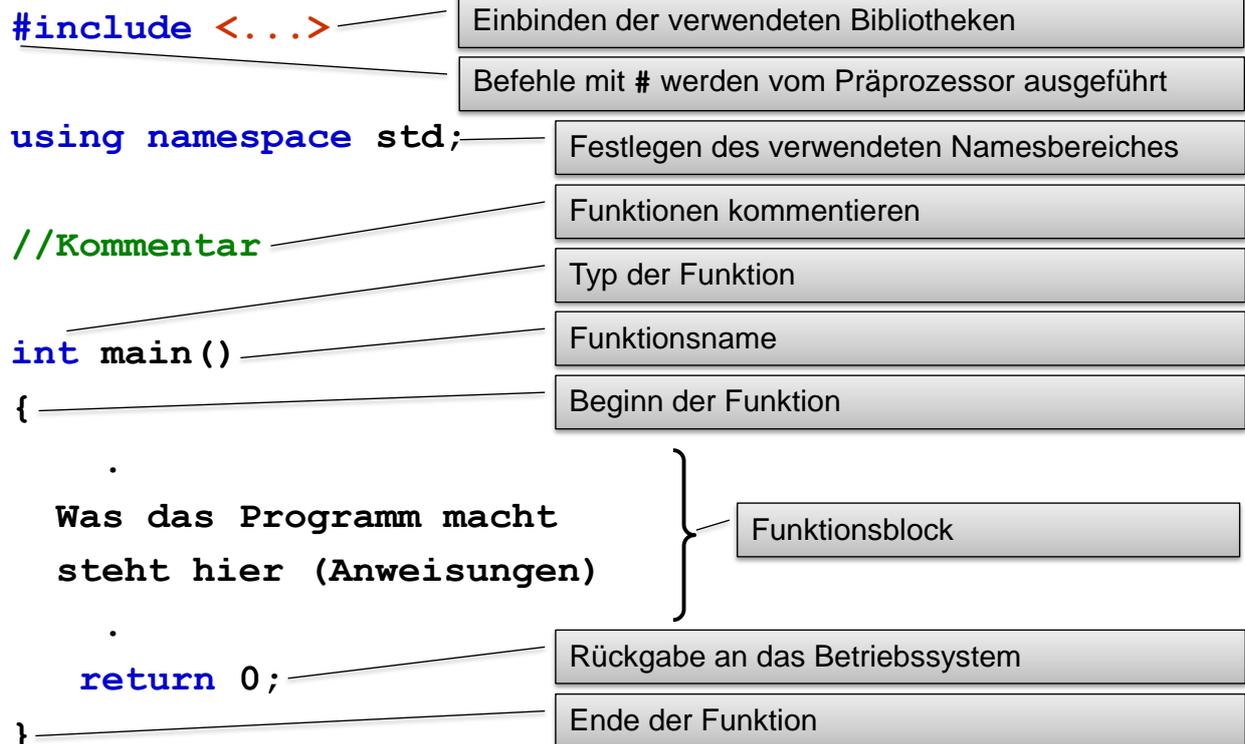
```

C:\> Eingabeaufforderung
D:\>cd code
D:\code>cd cpp
D:\code\cpp>cd hello
D:\code\cpp\hello>g++ -o HelloWorld.exe hello.cpp
D:\code\cpp\hello>HelloWorld.exe
Hello World
D:\code\cpp\hello>_
  
```

Escape-Sequenzen

Einzelzeichen	Bedeutung	ASCII-Wert (dezimal)
<code>\a</code>	alert (BEL)	7
<code>\b</code>	backspace (BS)	8
<code>\t</code>	horizontal tab (HT)	9
<code>\n</code>	line feed (LF)	10
<code>\v</code>	vertikal tab (VT)	11
<code>\f</code>	form feed (FF)	12
<code>\r</code>	carriage return (CR)	13
<code>\"</code>	"	34
<code>\'</code>	'	39
<code>\?</code>	?	63
<code>\\</code>	\	92
<code>\0</code>	Stringende-Zeichen	0
<code>\ooo</code> (bis zu drei Oktalziffern)	numerischer Wert eines Zeichens	ooo (oktal!)
<code>\xhh</code> (Folge von Hex-Ziffern)	numerischer Wert eines Zeichens	hh (hexadezimal!)

Aufbau eines Programms



Zwischenübung01: Prog. Aufbau

Ordnen Sie die folgenden Zeilen in der richtigen Reihenfolge, sodass sich ein lauffähiges Programm mit folgender Ausgabe ergibt



```

01 return 0;
02 cout << endl;
03 int main()
04 cout << "!!!" << endl;
05 {
06 //Dies ist die Hauptfunktion
07 #include <iostream>
08 cout << endl << "!!!";
09 cout << "\tMein zweites Programm";
10 using namespace std;
11 }
12 /* Die ganze Funktionalität dieses Programms */
13 cout << "\tMit einer zweiten Ausgabezeile" << endl;
14 cout << "!!!" << endl;
  
```



Zwischenübung01: Prog. Aufbau Lsg.

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

Ordnen Sie die folgenden Zeilen in der richtigen Reihenfolge, sodass sich ein lauffähiges Programm mit folgender Ausgabe ergibt

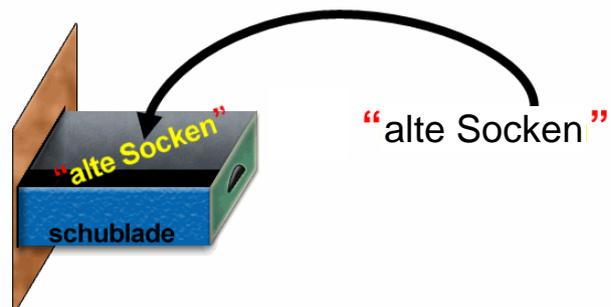


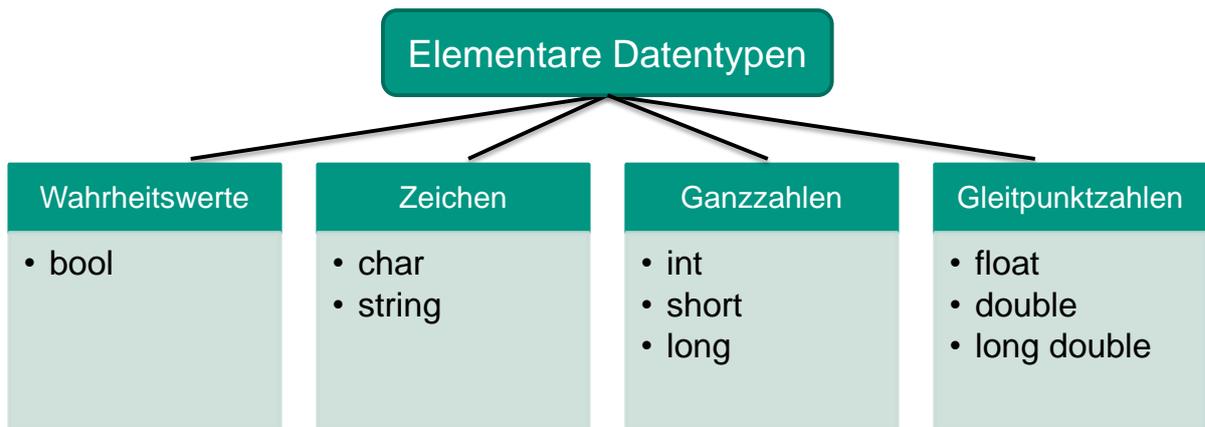
```
06 //Dies ist die Hauptfunktion
03 int main()
05 {
12     /* Die ganze Funktionalität dieses Programms */
08     cout << endl << "!!!";
02     cout << endl;
09     cout << "\tMein zweites Programm";
04/14 cout << "!!!" << endl;
13     cout << "\tMit einer zweiten Ausgabezeile" << endl;
14/04 cout << "!!!" << endl;
01     return 0;
11 }
```

```
C:\WINDOWS\system32\cmd.exe
?? ?
Mein zweites Programm!!!
Mit einer zweiten Ausgabezeile
?? ?
```

Daten speichern

- Problemstellung
 - Wie kann ich temporär Daten speichern?
 - Wie kann ich Eingaben des Benutzers temporär speichern?
- Lösung: *Variablen*
 - Unterscheiden sich nach Typ (Festpunkt, Gleitpunkt, Char, String, ...)
 - Unterscheiden sich in ihren Verwendungsmöglichkeiten
- Abstraktion
 - Verschiedene Schubladen
 - Unterschiedliche Form
 - Unterschiedliche Größe





- Ganzzahlen sind als **signed** (mit Vorzeichen) oder **unsigned** (ohne Vorzeichen) möglich



Typ	Speicherplatz	Wertebereich
Variablen für Wahrheitswerte		
bool	1 Bit	true oder false
Variablen für Zeichen		
char	1 Byte	-128 bis +127 bzw. 0 bis 255
Variablen für Ganzzahlen		
int	2 Byte bzw. 4 Byte	-32768 bis +32767 bzw. -2147483648 bis +2147483647
short	2 Byte	-32768 bis +32767
long	4 Byte	-2147483648 bis +2147483647
Variablen für Gleitpunktzahlen		
float	4 Byte	±3.4E+38 (Genauigkeit dezimal 6 Stellen)
double	8 Byte	±1.7E+308 (Genauigkeit dezimal 15 Stellen)

! Achtung, teilweise vom Compiler und von Bitbreite der Architektur abhängig!

Variablen Anlegen & Zuweisen

- Anlegen: `typ name1 [, name2, ...];`
 - Beispiel: `int zahl_x;`
 - Verwenden Sie aussagekräftige Namen

- Zuweisung eines Wertes mit „=“ Zeichen
 - Beispiel: `zahl_x = 5;`
`zahl_x = zahl_x + 1;`
 - Die rechte Seite wird der linken Seite zugewiesen
 - Variable repräsentiert danach diesen Wert

- Anlegen und Zuweisen in einem Schritt
 - Beispiel: `int zahl_y = 5;`



- Nichtinitialisierte lokale Variablen haben einen zufälligen Wert

Literale / Konstanten

- Literale
 - Feste Werte → unveränderlich im Programmcode hinterlegt
 - Beispiel: `cout << "Ich lerne C++" << endl;`
 - gibt "Ich lerne C++" aus / "Ich lerne C++" ist das Literal

- Konstanten
 - Variablen dessen Werte nicht mehr verändert werden können
 - Festlegen des Wertes beim Anlegen
 - Vorteil: Festlegen von Werten an zentraler Stelle
 - Schlüsselwort: `const`
 - Beispiel: `const int GESCHWINDIGKEIT = 100;`
 - `GESCHWINDIGKEIT` kann als normale Variable verwendet werden, ist allerdings unveränderbar

Eingabe von der Konsole lesen und speichern

- Einlesen von Benutzereingaben von der Konsole

- Beispiel:

```
int zahl;  
cin.sync();  
cin.clear();  
cin >> zahl;
```

- Vorbereiten von Eingaben

- `cin.sync();` ————— Puffer leeren

- `cin.clear();` ————— Fehlerflags löschen

- Eingabefunktion

- `cin` = console input

- `>>` = Zeichen von der Konsole in die Variable schieben

Zwischenübung02: Variablen

```
#include "iostream"
```

```
//Zwischenübung zu Variablen
```

```
int main()  
{  
    int zahl_x; zahl_y = 5;  
    float ergebnis = 123.45;  
    char buchstabe = 'A';  
  
    cout << Zahl_x << " " << Zahl_y << endl;  
    cout << zahl_y + 2 " " 2 * ergebnis << endl;  
  
    zahl_y = zahl_y + 5;  
    cout << zahl_y << " " << buchstabe << endl;  
  
    return 0  
}
```

Finden Sie die Fehler im
folgenden Programmcode
und bestimmen Sie die
Ausgabe.



Zwischenübung02: Variablen Lsg.

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

```
//Zwischenübung zu Variablen
```

```
int main()
{
    int zahl_x, zahl_y = 5;
    float ergebnis = 123.45;
    char buchstabe = 'A';

    cout << zahl_x << " " << zahl_y << endl;
    cout << zahl_y + 2 << " " << 2 * ergebnis << endl;

    zahl_y = zahl_y + 5;
    cout << zahl_y << " " << buchstabe << endl;

    return 0;
}
```

Finden Sie die Fehler im folgenden Programmcode und bestimmen Sie die Ausgabe.



```
C:\ d:Windows Eigene Dateien\Visual Studio 2005\
-858993460 5
7 246.9
10 A
```

Gültigkeitsbereiche

- Problemstellung:
 - Welche Variable existiert zu welchem Zeitpunkt?
 - Wo ist welche Variable sichtbar?
 - Wann kann ich auf welche Variable zugreifen?
- Lösung: Gültigkeitsbereiche
 - Bereich, in dem eine Variable / Objekt im Speicher existiert
 - Abhängig vom Ort und Art der Variablen
- Generelle Unterscheidung
 - Globale Variablen
 - Lokale Variablen
- **Regel:** Variablen immer so lokal wie möglich deklarieren

Globaler Gültigkeitsbereich (1)

- Globale Variable oder auch global deklarierte Variable
 - Variable wird bei Programmstart erzeugt, belegt während der ganzen Ausführungszeit Speicher, erst bei Programmende wieder gelöscht
 - Deklaration außerhalb von Funktionen

- Vorteil:
 - Variable kann überall in der gleichen Quelldatei genutzt werden

- Nachteile:
 - Variable belegt ständig Speicherplatz
 - Verlust der Übersicht bei größeren Programmen

- Verwendung globaler Variablen **minimieren** & begründen

Globaler Gültigkeitsbereich (2)

■ Beispiel:

```
[...]
int zahl;
void machWas ();

int main() {
    zahl = 25;
    machWas ();
    return 0;
}

void machWas () {
    cout << " globale Variable : " << zahl << endl ;
}
```

Diagramm zur Zuordnung von Codezeilen:

- Die Zeile `[...]` ist mit **Globale Variable** beschriftet.
- Die Zeilen `int zahl;` und `void machWas ();` sind mit **Funktionsdeklaration** beschriftet.
- Die Zeilen `int main() {` bis `}` sind mit **Hauptprogramm** beschriftet.
- Die Zeilen `void machWas () {` bis `}` sind mit **Funktionsdefinition** beschriftet.

Terminalausgabe:

```
C:\C:\Dokumente und Einstellungen\
globale Variable : 25
```

Lokaler Gültigkeitsbereich (1)

- Lokale Variable
 - wird in einem vom Blockoperator { } umrandeten Bereich deklariert
 - Kann sich innerhalb einer Funktionsdefinition, Schleife (oder einer anderen Kontrollstruktur) oder innerhalb einer Klasse befinden

- Existenz
 - Variable wird erst angelegt, wenn ihre Definition erreicht wird
 - lebt nur solange, wie sich der Programmablauf innerhalb der { }-Klammern befindet – außerhalb unbekannt / nicht verwendbar

- Vorteile:
 - Speicherplatz wird nur dann verbraucht, wenn er benötigt wird

- Nachteile:
 - Variablen / Werte müssen übergeben werden, wenn sie in anderen Funktionen verwendet werden sollen

Lokaler Gültigkeitsbereich (2)

■ Beispiel:

```

[... ]
void machWas ();

int main() {
  int zahl;
  zahl = 25;
  machWas ();
  zahl = 15;
  return 0;
}

void machWas () {
  cout << "globale Variable: " << zahl << endl;
}
  
```

Funktionsdeklaration

Hauptprogramm

Lokale Variable

Zugriff innerhalb der Funktion auf die lokale Variable möglich

Funktionsdefinition

Zugriff außerhalb der Funktion auf die Variable **nicht** möglich → Variable unbekannt

Syntaxfehler: Programm wird nicht kompiliert

Speicherklassen-Spezifizierer

- Statische Objekte = permanente Lebensdauer
 - Ändert nicht den Gültigkeitsbereich
 - Speicherbereich / Wert bleibt allerdings erhalten
 - Kennzeichnung durch das Schlüsselwort **static**

Beispiel: **static long summe;**

- Globale Objekte = Definition außerhalb einer Funktion
 - Informationsaustausch ohne Argumente
 - Überall in den Quelldateien eines Programms zugreifbar
 - Effekt auf das gesamte Programm → sparsam verwenden
 - Importieren von globalen Variablen aus anderen Quelldateien über den Speicherklassen-Spezifizierer **extern**

Beispiel: **extern long linie;**

Zwischenübung: Verdeckung

Was ist die Ausgabe des folgenden Programms?



```
[...]
int main() {
    int i = 100;

    for( int i = 0; i <= 5; i++ ) {
        int i = 1;
        i += 2;
        cout << i << ", ";
    }

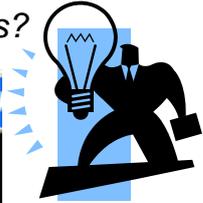
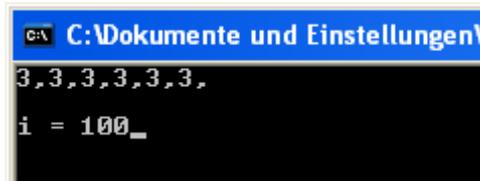
    cout << endl << endl;
    cout << "i = " << i;

    return 0;
}
```

Zwischenübung: Verdeckung Lsg.

Was ist die Ausgabe des folgenden Programms?

```
[...]  
int main() {  
    int i = 100;  
  
    for( int i = 0; i <= 5; i++ ) {  
        int i = 1;  
        i += 2;  
        cout << i << ", ";  
    }  
  
    cout << endl << endl;  
    cout << "i = " << i;  
  
    return 0;  
}
```



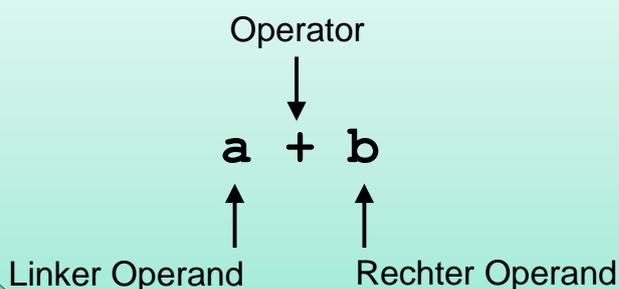
Tipp: Variablenüberdeckung vermeiden → verwenden von unterschiedlichen Variablennamen

Operatoren

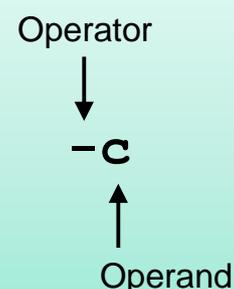
- Verarbeitung von Daten durch Operationen
- Operationen sind abhängig von der Art der Daten
 - z.B. keine Multiplikation mit nur einem Operanden
- Bei mehreren Operationen achten Sie auf die Priorität
 - Siehe C++ Kompendium für Prioritätenliste
 - Tipp: Klammern haben die höchste Priorität



Binäre Operatoren



Unäre Operatoren



Arithmetische Operatoren

Operator	Bezeichnung
+	Addition
-	Subtraktion
*	Multiplikation
/	Division
%	Modulodivision

Beispiel:

```
int a = 5;
int b = 3;
cout << a + b << endl;
cout << a - 5 << endl;
cout << 11 % 3 << endl;
```



```
cout << a / 0;
```

Ungültig, da Division durch Null nicht möglich

```
C:\WINDOWS\system32\cmd.exe
8
0
2
```

Unäre Arithmetische Operatoren

Operator	Bezeichnung
+ -	Vorzeichenoperator (unär)
++	Inkrement-Operator
--	Dekrement-Operator

Beispiel:

```
int c = 2; int d = 5;
cout << -d << endl;
d = c++;
cout << c << "\t" << d << endl;
d = ++c;
cout << c << "\t" << d << endl;
```

Keine Veränderung von d

c wird erst d zugewiesen, danach wird c inkrementiert

c wird erst inkrementiert und dann d zugewiesen

```
C:\WINDOWS\system32\cmd.exe
-5
3 4
4 4
```



```
cout << ++n << ++n;
```

Ungültig, da mehrere Inkrement- bzw. Dekrement Operationen mit der gleichen Variablen in einem Befehl nicht definiert sind

Operatoren für binäre Operationen

Operator	Bezeichnung
~	NICHT
&	UND
^	ODER, exklusives-ODER
<< >>	Links-Shift, Rechts-Shift

unsigned int a, b, c;	Bitmuster
a = 5;	00.....00000101
b = 12;	00.....00001100
c = ~a;	11.....11111010
c = a & b;	00.....00000100
c = a b;	00.....00001101
c = a ^ b;	00.....00001001
c = b << 3;	00.....01100000
c = b >> 2;	00.....00000011

Zuweisungsoperatoren

Operator	Bezeichnung
=	Einfache Zuweisung
op=	Zusammengesetzte Zuweisung (op) ist ein arithmetischer Operator

Beispiel:

```

int a = 2; int b = 3;
double c;
a = b + 3;
cout << a << endl;
b *= a;
cout << b << endl;

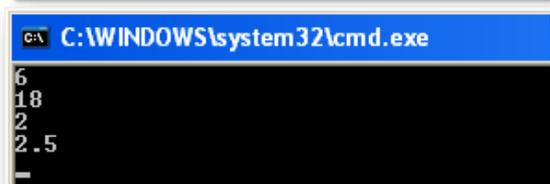
! c = 5 / 2;
  cout << c << endl;
  c = 5.0 / 2.0;
  cout << c << endl;
  
```

a wird verändert, b nicht

äquivalent `b = b * a;`

5 und 2 werden als `int` interpretiert

Interpretation als `double`



```

C:\WINDOWS\system32\cmd.exe
6
18
2
2.5
  
```

Operatorenpriorität

Priorität	Operator	Assoziativität
hoch	++ -- (Postfix)	
↑	++ -- (Präfix)	
	+ - (Vorzeichen)	
	! ~ (Nicht)	
	* / %	von links
	+ (Addition) , - (Subtraktion)	von links
	>> << (Rechts-, Links-Shift)	von links
	& (bitweises UND)	von links
	^ (bitweises exklusiv ODER)	von links
	(bitweises ODER)	von links
niedrig	Zuweisungsoperatoren: = += -= *= /= = <<= ...	von rechts

Assoziativität beschreibt die Reihenfolge der Auswertung

Zwischenübung03: Operatoren

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

```
//Zwischenübung zu Operatoren
```

```
int main()
```

```
{
```

```
    int zahl_x = 12, zahl_y = -2;
```

```
    cout << 10 / 3 << " " << zahl_x % 3 << endl;
```

```
    cout << 3 + 4 % 5 << " " << 3 * 4 % 5 << endl;
```

```
    zahl_x = -4 * zahl_y++ - 6 % 4;
```

```
    cout << zahl_x << " " << zahl_y << endl;
```

```
    zahl_x = zahl_x << 2;
```

```
    zahl_y = zahl_x & zahl_y;
```

```
    cout << zahl_x << " " << zahl_y << endl;
```

```
    return 0;
```

```
}
```

Bestimmen Sie die
Ausgabe des folgenden
Programms?



Zwischenübung03: Operatoren Lsg.



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

```
//Zwischenübung zu Operatoren
```

```
int main() {
    int zahl_x = 12, zahl_y = -2;
```

```
    cout << 10 / 3 << " " << zahl_x % 3 << endl;
    cout << 3 + ( 4 % 5 ) << " " << ( 3 * 4 ) % 5 << endl;
```

```
    zahl_x = ( ( -4 ) * zahl_y++ ) - ( 6 % 4 );
    cout << zahl_x << " " << zahl_y << endl;
```

postfix - wird als
letztes ausgeführt

```
    zahl_x = zahl_x << 2;
    zahl_y = zahl_x & zahl_y;
    cout << zahl_x << " " << zahl_y << endl;
    return 0;
}
```

Äquivalent

```
zahl_x = zahl_x * 4;
```

Bitmuster

```
zahl_y = 0xFFFFFFFF;
```

```
zahl_x = 0x00000018;
```

Referenz

- Kompendium: Kapitel 1 - Erste Schritte & Hello World!
Kapitel 2 - Variablen, Zeiger und Arrays (bis inkl. 2.8)
- Tutorium: Aufgabe 1 - 4

Arrays

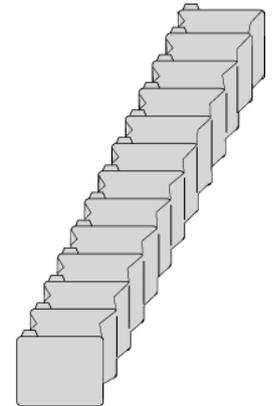
- Problemstellung:
 - Es sollen 100 Messwerte eines Experiments gespeichert werden
 - Anlegen von 100 Variablen des Typs `float` → viel Schreiarbeit ☹️
- Lösung: Arrays
 - Anlegen einer Vielzahl von Variablen des **gleichen** Datentyps
 - Unterscheidung über den Index

■ 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]	X 1234
zahlarr[1]	2
zahlarr[2]	X 554



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

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

```
double zahl[4][9];
zahl[0][2] = 9.65;
```

- Initialisierung bei Definition möglich
- Beispiel:

```
double 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

- Konvertierung des Wertes einer Variablen zu einer anderen Variablen mit unterschiedlichem Typ

- Beispiel:

```
int ganzzahl = 10;
double array[2];
double fliesskomma;
```

```
fliesskomma = ganzzahl;
```

```
ganzzahl = fliesskomma;
```

```
ganzzahl = ( int ) fliesskomma;
```

```
fliesskomma = array;
```

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

Fehler, 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

Zwischenübung04: Arrays

Definieren Sie ein Array, das ...



- das Monatsgehalt von 20 Angestellten speichern kann. Die ersten beiden Angestellten verdienen 3000.00 Euro.
- fünf Ganzzahlen speichern kann. Als Anfangswert erhält jedes Element das Doppelte seines Indexwertes.
- eine Ausgangsspannung in Abhängigkeit von 2 Widerstandswerten, 2 Eingangsspannungswerten und 3 Ausgangsstromwerten speichern kann.

Zwischenübung04: Arrays - Lsg.

Definieren Sie ein Array, das ...



- das Monatsgehalt von 20 Angestellten speichern kann. Die ersten beiden Angestellten verdienen 3000.00 Euro.
`double income[20] = { 3000.0, 3000.0 };`
- fünf Ganzzahlen speichern kann. Als Anfangswert erhält jedes Element das Doppelte seines Indexwertes.
`int twice[5] = { 0, 2, 4, 6, 8 };`
- eine Ausgangsspannung in Abhängigkeit von 2 Widerstandswerten, 2 Eingangsspannungswerten und 3 Ausgangsstromwerten speichern kann.

`double outputVoltage[2][2][3];`



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

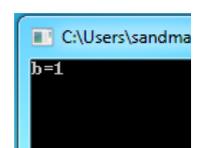
- Formulieren mit Vergleichsoperatoren
- Bedingungen verketteten mit logischen Operatoren
- Ergebnisse können in **bool** - Variablen gespeichert werden

- Es gilt:
 - 0 → **false**
 - alles andere → **true**

 - **false** → 0
 - **true** → 1

Beispiel:

```
bool b = false;
b = 3;
cout << "b=" << b << endl;
```



Bedingungen

- Beispiel: „Ist a mindestens so groß wie b ?“

- lautet in C++: `a >= b`

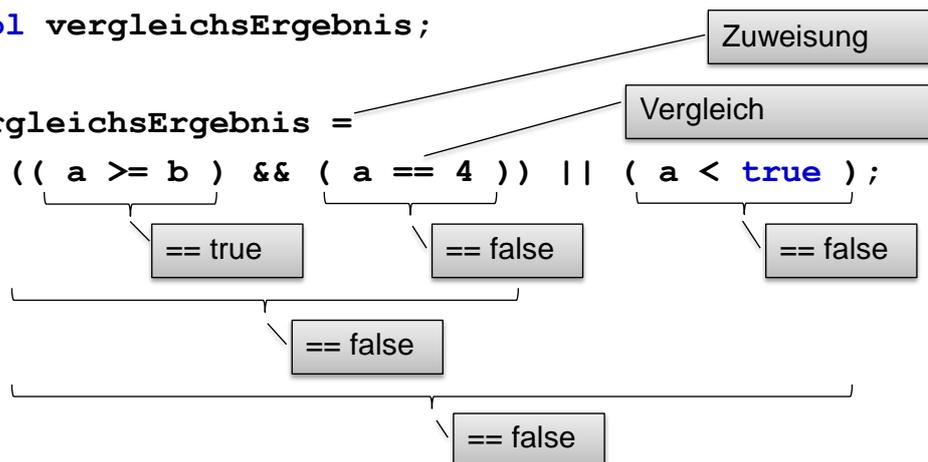
- Komplexeres Beispiel:

```
int a = 5, b = 3;
```

```
bool vergleichsErgebnis;
```

```
vergleichsErgebnis =
```

```
(( a >= b ) && ( a == 4 )) || ( a < true );
```



Vergleichsoperatoren

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

Beispiel:

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

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

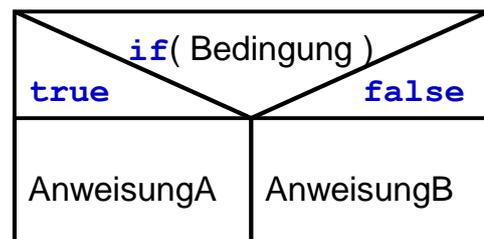
Verzweigungen

- Festlegen, welche Anweisung als nächstes ausgeführt wird

- Syntax:

```

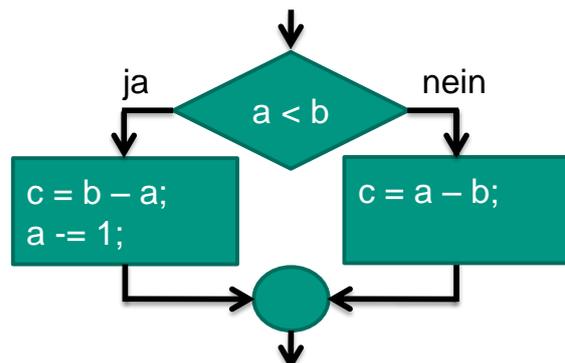
if( Bedingung ) {
    AnweisungA;
} else {
    AnweisungB;
} Optional
    
```



- else-Verzweigung ist optional

```

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



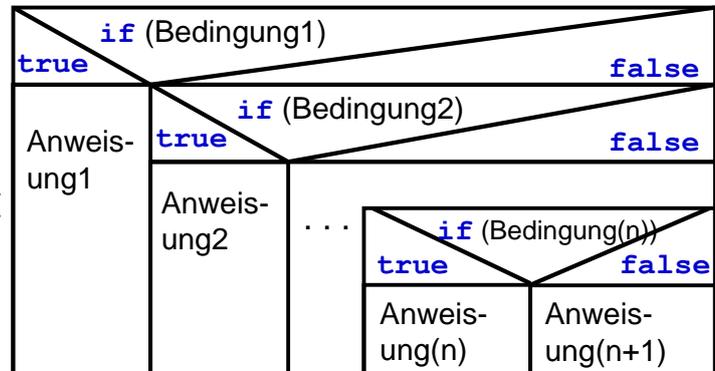
Mehrfachverzweigungen

- `if`-Verzweigungen können auch geschachtelt 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 );
}
  
```



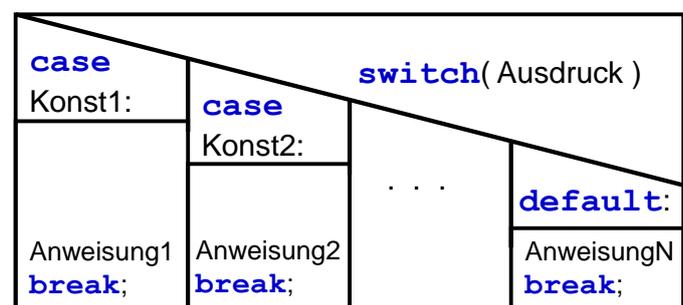
Mehrfachentscheidung

- 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();
        }
    }
}
}

switch( machineSt ) {
    case 1:
        continueWorking();
        break;
    case 0:
        startWorking();
        /* no break */
    case 2:
        calibrate();
        break;
    case -1:
        stopWorking();
        /* no break */
    default :
        callService();
}

```

Diese beiden
Programmabschnitte
erfüllen die gleiche Aufgabe!

Zwischenübung05: Verzweigungen

```

#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;
}

```

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



Zwischenübung05: Verzweigungen Lsg.

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

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

    if( a > b ) {
        betrag = a - b;
    } else {
        betrag = b - a;
    }

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

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



Wenn a größer als b ist
Dann ist $\text{betrag} = a - b$;
Ansonsten ist $\text{betrag} = b - a$;

```
C:\WINDOWS\system32\cmd.exe
a und b eingeben:12 56
|a-b| = 44

C:\WINDOWS\system32\cmd.exe
a und b eingeben:44 12
|a-b| = 32
```

Programmierrichtlinien (1)

- Problemstellung
 - Pflege von mehreren Programmen über einen längeren Zeitraum
 - Austausch von Programmcode unter mehreren Personen
 - Schreiben eines größeren Programms
- Lösung / Hilfestellung / Zu beachten dabei:
 - Schreiben von Programmen mit eindeutigen Schnittstellen
 - Beachten von Programmierrichtlinien
- Vorteile von Programmierrichtlinien
 - Besser verständliche Programme – vor allem für andere Entwickler
 - Vermeiden von Fehlern und Einfacheres Finden von Fehlern
 - Bessere Übersichtlichkeit und Wartbarkeit
- Außerdem Wichtig : KISS – Keep it simple and stupid

Programmierrichtlinien (2)

■ Schlechtes Beispiel:

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

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

if (a > b) betrag = a - b;
else
betrag = b - a;
cout << " |a-b| = " << betrag
<< endl;
return 0; }
```

Ansicht des
übersichtlichen
Programmcodes s.
Zw.Übung 5

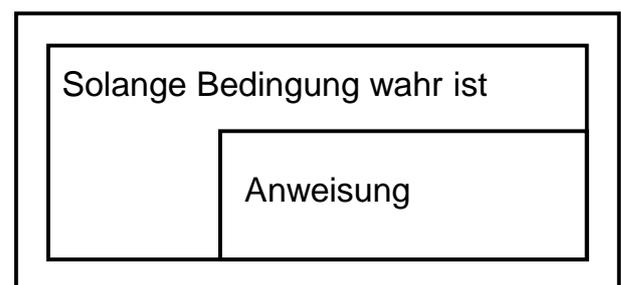
While - Schleife

- Viele Sachen müssen mehrfach wiederholt werden
 - Feste Anzahl Wiederholungen oder abhängig von einer Bedingung

■ while-Schleife - Kopfgesteuert

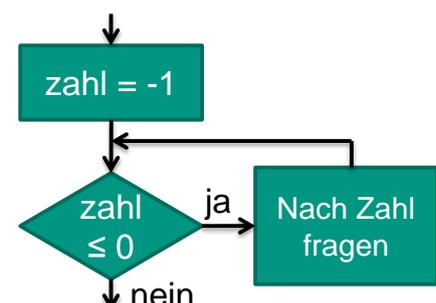
- Bedingung **true** oder **false**

■ Syntax: **while** (Bedingung) {
 Anweisung
}



■ Beispiel:

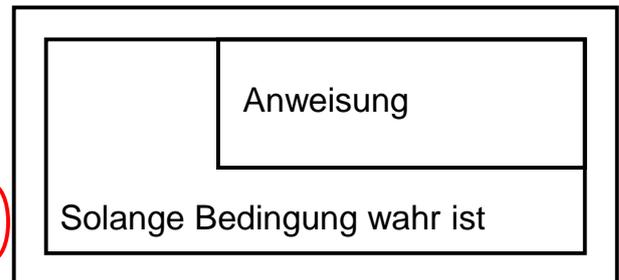
```
int zahl = -1;
while( zahl <= 0 ) {
    cout << "Bitte geben Sie eine";
    cout << "Zahl größer Null ein: ";
    cin >> zahl;
}
```



Do-While - Schleife

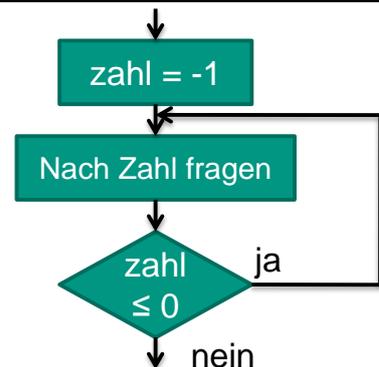
- Auch Fußgesteuerte Schleife genannt
 - Wird mindestens einmal aufgeführt (Unterschied zur **while**-Schleife)

- 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 );
```

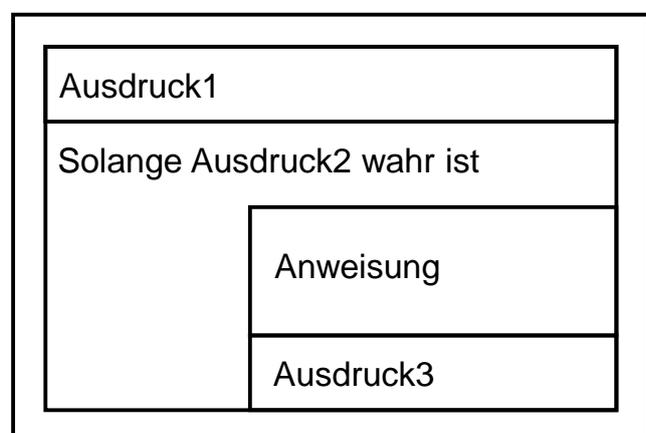


For – Schleife (1)

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

- Syntax: `for(Ausdruck1; Ausdruck2; Ausdruck3) {`
 Anweisung;
 `}`

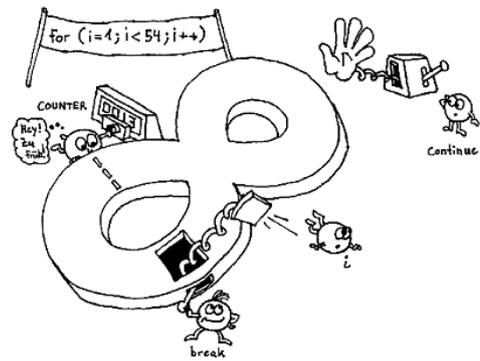
- Ausdruck1 = Initialisierung
 - Zu Beginn der Schleife einmalig ausgeführt
- Ausdruck2 = Bedingung
 - Schleife läuft, solange Bedingung wahr
- Ausdruck3 = Veränderung
 - Am Ende jedes Schleifen-Durchlaufs ausgeführt



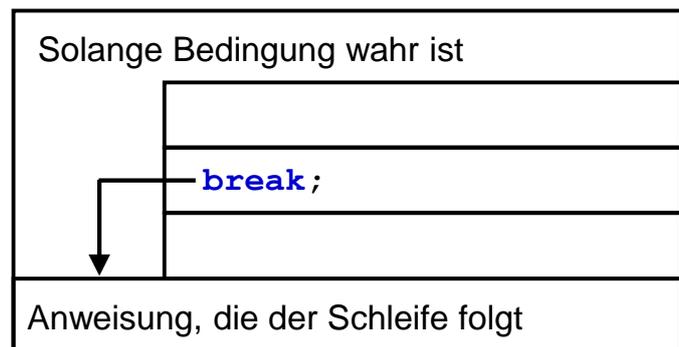
For – Schleife (2) und break;

■ 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



Zwischenübung06: Schleifen

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

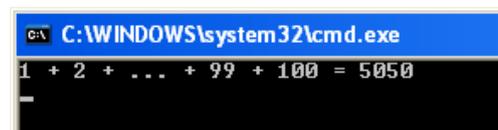
```
int main() {
    unsigned long sum = 0;
```

```
// Hier ergänzen
```

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

```
return 0;
}
```

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



Zwischenübung06: Schleifen Lsg.

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

```
int main() {
    unsigned long sum = 0;

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

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

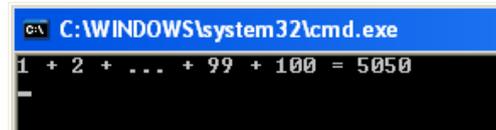
    return 0;
}
```

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



Oder:

```
int i = 1;
while( i <= 100 ) {
    sum += i;
    i++;
}
```



```
C:\WINDOWS\system32\cmd.exe
1 + 2 + ... + 99 + 100 = 5050
```

Referenz

- Kompendium: Kapitel 2 – Variablen und Arrays (ab 2.9)
Kapitel 3 – Kontrollstrukturen (komplett)