Einführung in die Softwareentwicklung

SOMMERSEMESTER 2020

```
// This is C++
void main() {
    int var = 42;
    std::string s = "Hello World!\n";
    s += "The answer is still: ";
    s += string(var);
    std::cout << s;
}
```

Foliensatz #02

Programmstrukturen

Michael Wand · Institut für Informatik · michael.wand@uni-mainz.de

C++ Syntax

Syntax

File: HelloWorld.cpp

```
void func(int value) {
    // print value
    cout << value;
}
```

Whitespaces

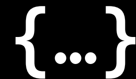
- „Whitespaces“ trennen Elemente
  - Leerzeichen, Tabs, Returns
- Sonderzeichen (Klammern, Punkte, etc.) trennen auch ohne Whitespaces

```
Console.out.write("Hello");
```
=
```
Console . out .   write ( "Hello" ) ;
```

Regeln

File: HelloWorld.java

```
void sendGreetings(int a) {
  if (a == 42) {
      // print some greetings
      cout << "Hello World!";
   }
}
```



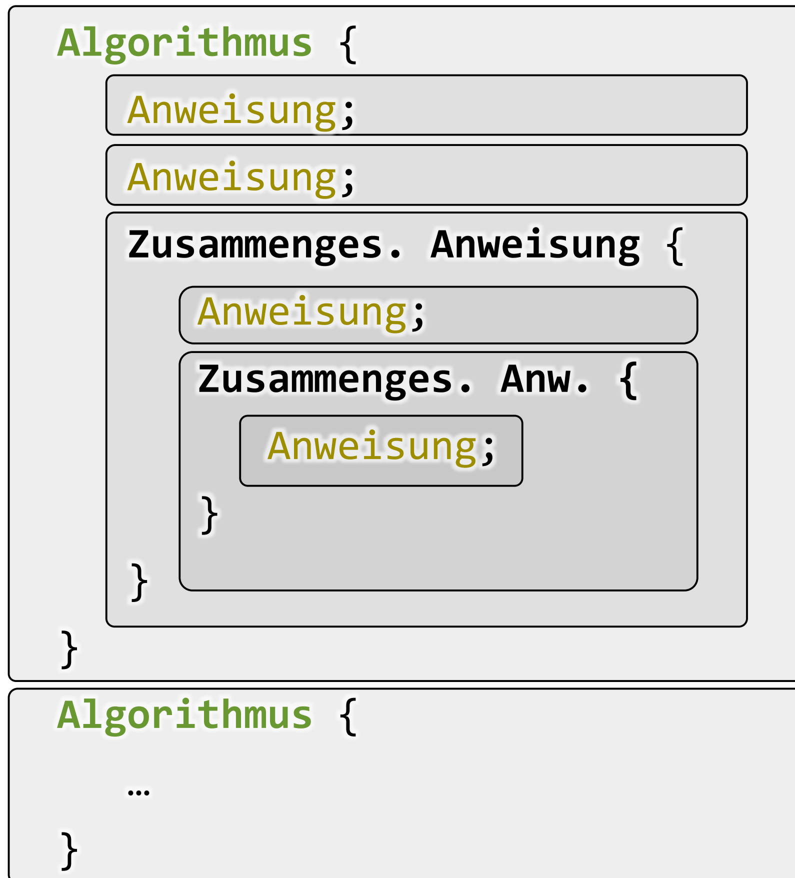
Blockstruktur

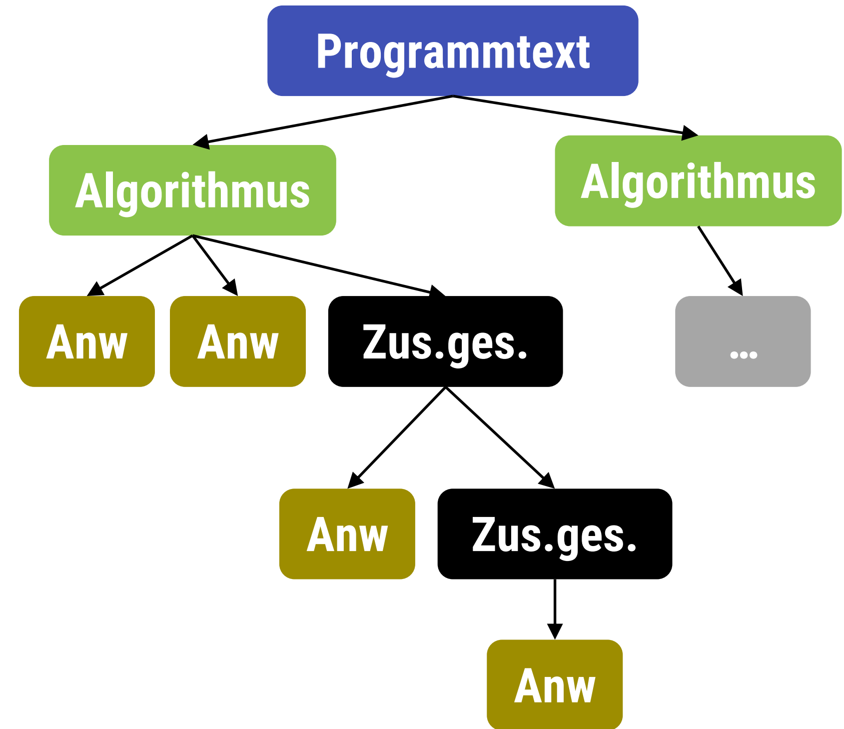
- Programm besteht aus geschachtelten Blöcken
- Konvention: Einrücken
  - Anders als in Python – Einrückung hat keine Bedeutung!

Schachtelung

```
Algorithmus {
    Anweisung;
    Anweisung;
    Zusammenges. Anweisung {
        Anweisung;
        Zusammenges. Anw. {
            Anweisung;
        }
    }
}
Algorithmus {
    …
}
```

Schachtelung





„Baumstruktur"

Hierarchische Schachtelung

„Divide & Conquer"

Geschmackssache

Variante 1 (K&R, Stoustrup, auch üblich in JAVA)

```
void something(int a) {
   // print some greetings
   if (a == 42) {
      cout << "Hello World!" << "\n" << "The number is" << a;
   }
}
```

Variante 2 (auch beliebt)

```
void something(int a)
{
   // print some greetings
   if (a == 42)
   {
      cout << "Hello World!" << "\n" << "The number is" << a;
   }
}
```

Regeln

File: HelloWorld.cpp

```
void something(int a){/*print some greetings*/if(a==42){cout
<<"Hello World!"<<"\n"<<"The number is"<<a;}}
```

Einrückung

- Reine Konvention
  - Gleiches Ergebnis ohne
  - Anzahl Whitespaces irrelevant
  - Lesbarkeit wichtig! (human factors)
- Farben („Syntaxhighlighting“) hier nur zur Illustration

C++

Kontrollfluss

Kontrollfluss in Python

Sequenz

```
# nacheinander ausführen
x = 42
y = 23
x = x + y
print(x)
```

Wiederholung

```
# Beispielschleife
x = 23
while x <= 42:
    print(x)
    x = x + 1
```

Fallunterscheidung

```
# Bsp.-Fallunterscheidung
if x == 42:
    print("very meaningful")
else:
    print("don‘t care")
```

Unterprogramme

```
# Unterprogram
def comp_sum(x, y):
    z = x + y
    return(z)
```

Kontrollfluss in C++

Sequenz

```
// nacheinander ausführen
int x = 42;
int y = 23;
x = x + y;
cout << x;
```

Wiederholung

```
// Beispielschleife
x = 23;
while (x <= 42) {
    cout << x;
    x = x + 1;
}
```

Fallunterscheidung

```
// Bsp.-Fallunterscheidung
if (x == 42) {
    cout << "very meaningful";
} else {
    cout << "don‘t care";
}
```

Unterprogramme

```
// Unterprogram
int compSum(int x, int y)
{
    int z = x + y;
    return z;
}
```

C++

Fallunterscheidungen

Fallunterscheidungen in C++

Beispiel

Fallunterscheidung

```
int variable;
...
if (variable == 42) {
   // Ausgeführt, falls Bedingung wahr
   cout << "The meaning of life.";
} else {
   // Ausgeführt, falls Bedingung falsch
   cout << "The number is not 42.";
}
```

Struktogramm:

Falls variable = 42	
	Ausgabe „The meaning of life“
Sonst	
	Ausgabe „The number is not 42“

Muster

Allgemeines Muster

- ```
  if ( <logischer Ausdruck> ) {
     <Anweisungen1>
  } else {
    <Anweisungen2>
  }
  ```
- Der logische Ausdruck muß einen Wahrheitswert zurückliefern.
  - Falls das Ergebnis TRUE ist, werden <Anweisungen1> ausgeführt, sonst <Anweisungen2>
  - Die Anweisungen können beliebig komplex sein (Sequenz von Anweisungen, oder auch geschachtelte Blöcke).

  ```

# Muster

## Spezialfall

- ```
  if ( <logischer Ausdruck> ) {
      <Anweisungen1>
  }
  ```
- Der „**`else`**“ Teil kann ausgelassen werden.

  ```

# Muster

## Spezialfall

- ```
  if ( <logischer Ausdruck> )
      Einzelne_Anweisung;
  ```
- Wenn nur eine Anweisung folgt, können die Klammern weggelassen werden
  - Nicht empfohlen – fehleranfällig und unübersichtlich.

Logik dahinter

- Es folgt immer eine Anweisung
- Mittels „{…}“ können mehrere Anweisungen zu einer zusammengefaßt werden.

  ```

# Achtung!

## „Dangeling else“:

- Mehrdeutigkeiten möglich

```
if (a == 1)
    if (b == 1)
        c = 2;
    else
        c = 3;
```

Korrekt!
Binden an innerstes „if“

```
if (a == 1)
    if (b == 1)
        c = 2;
else
    c = 3;
```

Falsch
Einrückung ≠ Ausführung

Kaskaden

Sinnvoller Spezialfall: Kaskade von Fällen

```
if (a == 1) {
   cout << "option 1";
} else if (a == 2) {
   cout << "option 2";
} else if (a == 3) {
   cout << "option 3";
} else {
   cout << “not 1-3";
}
```

Kaskaden

Bemerkungen

- Ratschlag: Immer geschweifte Klammern benutzen (außer bei Kaskaden)
- Kaskaden können auch mit „`switch`“ realisiert werden (siehe Lehrbücher / Google-it)

Algorithmische Bausteine

Sequenz

Baustein 1
Baustein 2
...
Baustein n

Fallunterscheidung

Falls <Bedingung>	
	<Baustein 1>
Sonst	
	<Baustein 2>

Wiederholung

Wiederhole	
	<Baustein>
Solange, bis <Bedingung>	

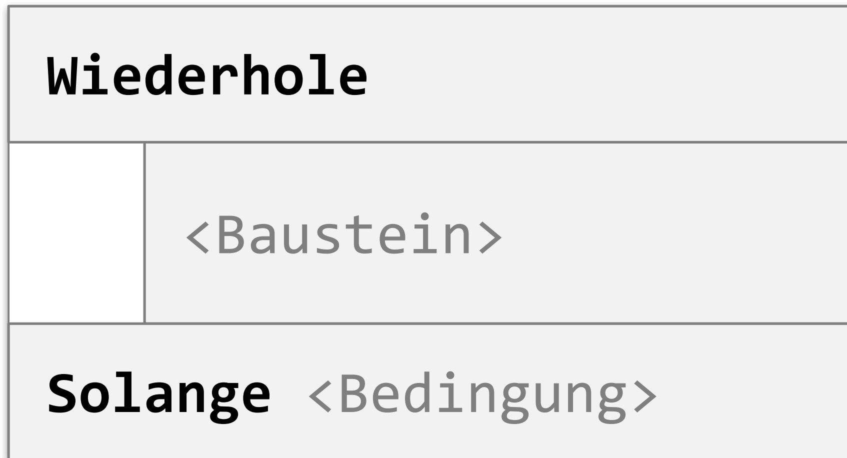
Unterprogramme

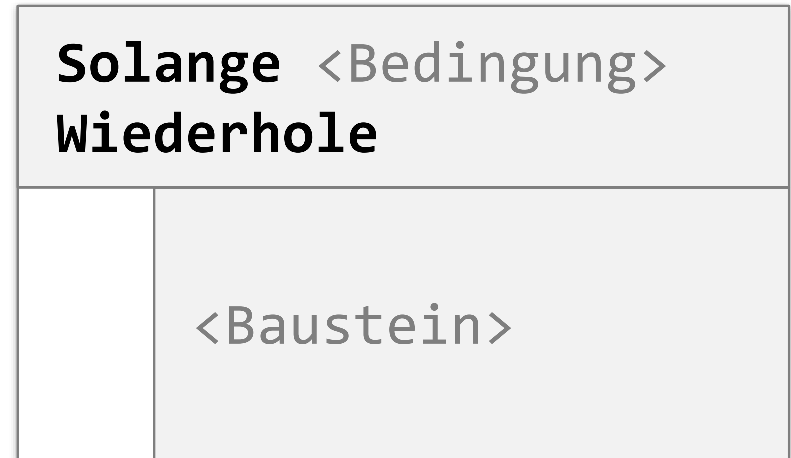
```
...
Name( Eingaben    ← <Ausdruck>,
      <Variable> → Ausgaben    )
...
```

Unteralgorithmus: <Name>
Eingabe: <Variable> Ausgabe: <Variable>
<Baustein>

Struktogramme
Wiederholungen

Wiederholung





Bedingung am Ende überprüfen

Bedingung am Anfang überprüfen

hier: unterschiedliche Semantik!

Wiederholung von Bausteinen

- Baustein wiederholt ausführen
- Ende wenn Bedingung nicht mehr erfüllt ist

C++

Wiederholungen

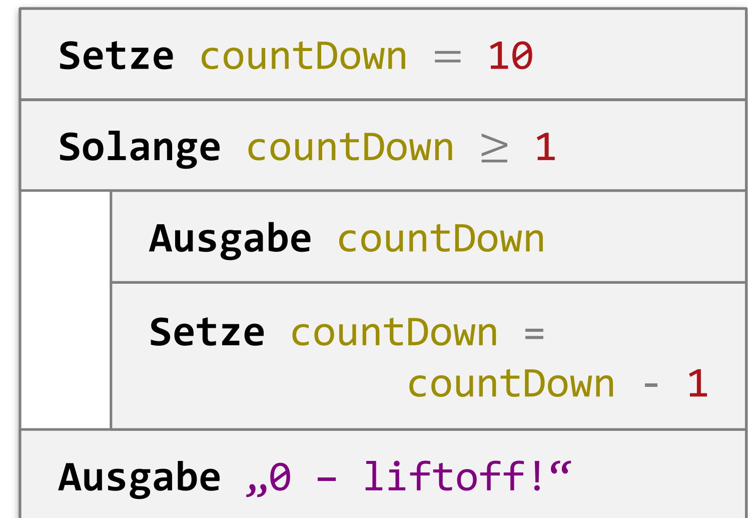
Wiederholungen in JAVA

Beispiel

Wiederholung

```
int countDown = 10;
while (countDown >= 1) {
    cout << countDown << ", ";
    countDown--;
}
cout << "0 – liftoff!";
```

Struktogramm:



Ausgabe

```
> myProg.exe
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 – liftoff!
>
```

Muster

Allgemeines Muster

```
while ( <logischer Ausdruck> ) {
    <Anweisungen>
}
```

- Logischer Ausdruck liefert Wahrheitswert.
  - Falls er zutrifft, werden die Anweisungen ausgeführt.
  - Danach wird wieder überprüft, usw.
- Achtung: Überprüfung am Anfang!

Ohne Klammern

Variante

```
while ( <logischer Ausdruck> )
    <Anweisungen>;
```

- Einzelne Anweisung statt {…} möglich.
- Nicht empfohlen

Muster

Überprüfung am Ende

```
do {
    <Anweisungen>
} while ( <logischer Ausdruck> )
```

- Anweisungen werden immer einmal ausgeführt
- Danach wird logischer Ausdruck berechnet.
  - Falls er zutrifft: Wiederholung (Sprung zurück auf „do“)
  - Sonst Ende der Schleife
- Achtung: Überprüfung am Ende!

break / continue

Vorzeitiger Abbruch

```
while ( <logischer Ausdruck> ) {
    ...
    break;
    ...
}
```

- Break verläßt Schleife
  - Sprung ans ende (nach „}“)

break / continue

Encore une fois!

```
while ( <logischer Ausdruck> ) {
    ...
    continue;
    ...
}
```

- „continue“ geht direkt zurück an Anfang
  - Rest des Schleifenrumpfs wird ignoriert
  - Direkt an Anfang: Überprüfung der Schleifenbedingung

Beispiel

Endlosschleife

```
while ( true ) {
    <Anweisungen>
}
```

- Schleife endet nie – Absturz eingebaut?
- Erlaubt
  - z.B. Event-loops für Webserver
  - Break erlaubt trotzdem Ausstieg.

Muster

Spezialfall

```
for (<init>; <Bedingung>; <amEnde>) {
    <Anweisungen>
}
```

- „for“ Schleife – wichtigster Spezialfall
  - Eigener Befehl dafür
  - Syntax ist etwas *schrill*
  - Übernommen aus C *(C ist ziemlich cool)*

Muster

Spezialfall

```
for (<init>; <Bedingung>; <amEnde>) {
    <Anweisungen>
}
```

- Äquivalent zu:

```
<init>;
while (<Bedingung>) {
    <Anweisungen>;
    <amEnde>;
}
```

100% Äquivalent:

for-Schleife

```
for (int countDown = 10; countDown >= 1; countDown--) {
    cout << countDown;
    cout << ", ";
}
cout << "0 - liftoff!";
```

while-Schleife

```
{int countDown = 10;
while (countDown >= 1) {
    cout << countDown;
    cout << ", ";
    countDown--;
}}
cout << "0 - liftoff!";
```

C++

Unterprogramme

Unterprogramme

File: HelloWorld.cpp

```
#include <iostream>
using std::cout;

void main() {
   // print some greetings
   cout << "Hello World!";
}
```

Einfach: Weitere Funktionen hinzufügen

Beispiel

File: HelloWorld.cpp

```
#include <iostream>
using std::cout;

void greetings() {
    // print some greetings
    cout << "Hello World!";
}

void main() {
    // Aufruf des Unterprogramms
    greetings();
}
```

Beispiel

File: ArithFunc.cpp

```
#include <iostream>
using std::cout;

int addNumbers(int a, int b) {
   int c = a + b;
   return c;
}
void main() {
      cout << "23 + 42 ergibt: ";
      // Aufruf des Unterprogramms
      cout << addNumbers(23, 42);
}
```

Allgemeines Muster

Parameterliste (Eingabe)
mit Typ

```
int addNumbers(int a, int b) {…}
```

Ausgabe
(Rückgabetyp)

Name (Bezeichner)

Funktionsdeklaration

- Benutzergewählter Bezeichner als Name
- Eingabeparameter: (Typ1 Name1, Typ2 Name2, …)
- Nur eine Ausgabe! (Typ angeben)
  - Ausgabe mit „return“ Anweisung
  - Kann jederzeit ausgeführt werden – beendet Unterprogramm

„return“

Rückgabe von Werten mit „return“

- Jederzeit möglich
  - Unterprogram wird beendet
- Bei „void“ Rückgabe: einfach „return;“
  - Direktes Ende des Unterprogramms
  - Rückkehr zum Aufrufer

Lokale Variablen

Variablen in Unterprogrammen

- C++ Funktionen haben immer „lokale Variablen“
  - Nicht nach außen sichtbar
  - Eigener Speicherplatz (auch bei gleichem Namen!)
- Speicherplatz
  - Wird beim Aufruf angelegt
    - Für alle lokalen Variablen auf einmal
  - Wird beim Verlassen wieder freigegeben
- Eingabeparameter sind Kopien in lokale Variablen!
- Ausgabe (return) auch per Kopie!
- C++ ist immer „Call-by-Value“ (Kopie des Wertes)

Lokale Variablen

Lokale Variablen

```
int addNumbers(int a, int b) {
    int c = a + b;
    return c;
}
void main() {
    int c = 123;
    // Aufruf des Unterprogramms
    int d = addNumbers(1,2);
}
```

Kein Einfluß! verschiedene Variablen!

Lokale Variablen

Lokale Variablen

```
int addNumbers(int a, int b) {
    int c = a + b;
    return c;
}

void main() {
    // Aufruf des Unterprogramms
    int d = addNumbers(1,2);
    d = d + c;
}
```

Fehler!
c ist hier unbekannt

Lokale Variablen

Lokale Variablen

```
int addNumbers(int a, int b) {
    int c = a + b;
    return c;
}
void main() {
    int a = 42; int b = 23;
    // Aufruf des Unterprogramms
    int d = addNumbers(1,2);
    // Ergebnis d == 3 (nicht 65)
    // Werte von a,b weiterhin 42,23
}
```

Kein Einfluß!
verschiedene
Variablen!

„Const“-Paramter

Lokale Variablen

```
// „Const“-Parameter können nicht mehr geändert werden
// Erlaubt Compiler mehr Optimierungen
// und vermeidet Fehler
int addNumbers(const int a, const int b) {
    int c = a + b;
    a += 42; // nicht erlaubt – Compilerfehler!
    return c;
}
void main() {
    int d = addNumbers(1,2);
}
```

Referenzparameter

Lokale Variablen

```
// Referenzzarameter <Typ>& werden als Referenzen übergeben
// Erlaubt Ein- und Ausgabe (beides möglich)
// Äußere Variable wird an Stelle von result verwendet (und ggf. verändert)
void addNumbers(const int a, const int b, int &result) {
    result = a + b;
    a += 42;  // nicht erlaubt – Compilerfehler!
}
void main() {
    int theResult;
    addNumbers(1, 2, theResult);  // letztes Arg. muss eine Variable sein!
    cout << "1+2 ist " << theResult;
    addNumbers(1, 2, 3);  // Compilerfehler! Referenzparameter keine Variable
}
```

Überladen (nur C++)

Lokale Variablen

```
// Verschiedene Parameterlisten möglich;
// Auswahl / Unterscheidung durch Aufruftyp (Rückgabe nicht betrachtet)
int addNumbers(int a, int b) {
    return a + b;
}
double addNumbers(double a, double b) {
    return a + b;
}
void main() {
    int c = addNumbers(1, 2);               // erste Variante
    double d = addNumbers(1.0, 2.0);        // zweite Variante
    double e = addNumbers(1, 2.0);          // Compilerfehler - mehrdeutig
    double e = addNumbers(1.0f, 2.0f);      // Compilerfehler - mehrdeutig
}
```