

Kapitel 1: Einstieg in C++ und Objektorientierung

Alles ganz ähnlich wie C... oder?!

```
int main(int argc, char *argv[]) {  
    return 0;  
}
```

Dies ist auch gültige C++-Code!

- C-Programme sind größtenteils gültige C++-Programme (gleiche Codebasis)
- z.B. Funktionen, native Typen (int, float, void...),
main(), Pointer, Arrays
... alles wie in C.

Alles ganz anders als in C... oder?!

wie C
`#include <iostream>`

Konvention: kein .h mehr bei Dateinamen von Header-Dateien

wie C
`int main() {`

Name = space
`std::cout`

Scope-Operator
`::`

cout - Objekt: Output Stream
↳ Datenstrom

return 0;

wie C
`}`

Zeilenende
`std::endl;`

`printf()` könnte man nach wie vor verwenden!

cout? Streams!

cout (vollständige Name: `std::cout`)

- Ein *Objekt*, das in `<iostream>` definiert ist
- Ein Output Stream, der mit `stdout` verknüpft ist
- `std::cout << Inhalt` $\hat{=}$ `printf(Inhalt)` in C
- `std::cerr` $\hat{=}$ `stderr` in C

Analog:

- Output Streams in Dateien C: `fprintf()`
- Output Streams in Speicher (RAM) C: `fprintf`, Strings füllen!

Idee: Datenstrom "Hello World" $\xrightarrow[\text{Stream}]{\text{cout}}$ `ffg(Bildschirm)`

std::? Namespaces!

In C:

- globale Variablen
- lokale Variablen

↳ „global global“
↳ global static (Datei-Scope)

→ Scope (Gültigkeit, Sichtbarkeit)

C++ führt *Namespaces* ein:

künstliche, vom Programmierer erfundene Scope

Bereits vorgegeben: `namespace std { ... }`

↳ Namen darin müssen jetzt mit `std::` bezeichnet werden, um sichtbar zu sein

`std` ist für Namen von Standard-C++ reserviert

Left-Shift eines Strings? Operatoren!

`cout << 1` bedeutet nicht shift left!

`3 << 1` also schon (es gibt in C++ `<<` auch noch
als Bit-Shift)

⇒ Operatoren wie `<<` sind in C++ (anders als in C) *Funktionen*

Konkreter Name hier: `operator<<()` "Stream Insertion Operator"

→ `+` `-` `*` `/` `%` ...

⇒ `cout << 1` $\hat{=}$ `cout.operator<<(1)` *Seides korrekt!*

"Führe am Objekt `cout` die Funktion `operator<<()` aus"

2. Beispiel: Eingabe

```
#include <iostream>
```

```
int main() {
```

```
    int a;
```

```
    std::cin >> a;
```

```
    std::cout << "a is " << a << std::endl;
```


```
}
```

Stream extraction operator
 $\hat{=}$ fscanf(stdin, ...)

Stream insertion op.

Wof aus stdin (z.B. Tastatur) landet in Variable a

Sichtweise des Designs: Kaffeemaschine

- Prozedural: 



Maschine, die Algorithmus zum Kaffeekochen abarbeitet

- Objektorientiert 

Maschine, die "weiß", wie man Kaffee kocht

⇒ Kaffe Zubereitungswissen ist verborgen, "gekapselt"

Kaffeemaschinen-"Objekt":

- Enthält Kaffeepulver, Wasser 
- Kann daraus Kaffee zubereiten 

Umsetzung in den Programmiersprachen: C

```
struct Turtle {int years; ...};
```

Daten / Eigenschaften

```
void increaseAge(struct turtle turtle, int yearsToAdd) {  
    turtle1.years += yearsToAdd  
}
```

Funktion modifiziert Daten des struct

```
struct Turtle turtle1;  
increaseAge(turtle1, 10);
```

- Funktionsaufruf erhält alle benötigten Daten von außen
- Daten sind anschließend verändert *sichtbar*

Umsetzung in den Programmiersprachen: C++

↗ ~ struct

```
class Turtle {  
public:  
    int years; Daten  
    void increaseAge(int yearsToAdd) {  
        this.years += yearsToAdd;  
    }  
    ...  
};
```

→ "Methode" ist Bestandteil der Klasse

⇓
Klasse "kennt" alle Daten der Turtle

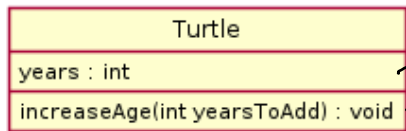
```
Turtle turtle1;
```

```
turtle1.increaseAge(10);
```

↪ Aufruf der Methode am Turtle-Objekt

- Daten sind gekapselt
- Aufruf enthält nur zusätzliche Werte

1. UML-Beispiel: Grafische Darstellung von Klassen



Daten - Compartment
Methoden - "

In PlantUML:

```
@startuml
class Turtle {
    years : int
    increaseAge(int yearsToAdd) : void
}
hide circle
@enduml
```