

Inhalt

- Vorstellung
- Einleitung, Geschichte
- Grundlagen
- **Klassen**
- Vererbung
- Zusätzliche Themen
- Die Standardbibliothek

Klassen

- Klassentypen
- Konstruktor / Destruktor
- Copy und Move Konstruktor
- Überladen von Operatoren
- Statische Variablen und Methoden
- Dynamische Speicherallokation

Klassen

Klassentypen

- Sind Datentypen, die Attribute (Variablen) und Elementfunktion in Objekten zusammenfassen
 - → struct und class
- Zugriffsrechte:
 - public: freier Zugriff von außen erlaub
 - private: Zugriff nur in Methoden der Klasse
 - protected: Zugriff nur in Methoden von verwandten Klassen
- Struct ist standard public
- Class ist standard private
- Definition des structs Komplex:

```
struct Komplex {  
    double _re; // Membervariable  
    double _im;  
    void ausgabe(); // Methode  
};
```

Klassen

Klassentypen

- Implementierung der Methode `Komplex::ausgabe()`
- this: Zeiger auf das Exemplar, dessen Methode ausgeführt wird, man diesen auch weglassen, dann wird implizit auf die Membervariablen zugegriffen → **Vorsicht mit Namenskonflikten bei Funktionsparametern!**

```
void Komplex::ausgabe() {  
    cout << '(' << _re << ',' << _im << ')' << endl;  
}
```

```
void Komplex::ausgabe() {  
    cout << '(' << this->_re << ',' << this->_im << ')' << endl;  
}
```

Klassen

Klassentypen

- Der Datentyp "Komplex" kann auch als Klasse dargestellt werden. Um von außen darauf zuzugreifen müssen diese als public deklariert werden.
- Typischerweise sind Membervariablen private und deren Methoden public

```
class Komplex {  
public:  
    double _re; // Membervariable  
    double _im;  
    void ausgabe(); // Methode  
};
```

```
class Komplex {  
    double _re; // Membervariable  
    double _im;  
public:  
    void ausgabe(); // Methode  
};
```

Klassen

Konstruktor / Destruktor

- Konstruktor: Spezielle Methode die automatisch ausgeführt wird nachdem eine Instanz/Objekt der Klasse initiiert wurde.
 - → Zweck: Initialisieren der Member

```
cKomplex::cKomplex() {  
    _re = _im = 0;  
}
```

- Initialisierungsliste:

```
cKomplex::cKomplex(): _re(0), _im(0) { }
```

```
class cKomplex {  
    double _re; // Membervariable  
    double _im;  
public:  
    cKomplex();  
    cKomplex(int re, int im);  
    ~cKomplex();  
    void ausgabe(); // Methode  
};
```

- Destruktor: Automatischer Aufruf kurz vor Zerstörung.
 - → Zweck: Zusätzlich belegte Ressourcen freigeben.

```
cKomplex::~~cKomplex() {  
    cout << "Bye, bye!" << endl;  
}
```

Klassen

Überladen von Operatoren

- Überladen von rationalen und logischen Operatoren
 - → Ergebnis der Operation muss ein ganzzahliger *true*- oder *false*-Wert sein

```
int cKomplex::operator==(const cKomplex obj) {  
    if (_re == obj._re && _im == obj._im) {return true;}  
    else {return false;}  
}
```

- Überladen von unären Operatoren

```
++obj;  
// Hier wird die Operatorfunktion ohne Parameter definiert.  
cKomplex operator++();  
// ---  
obj++;  
// Zur Unterscheidung wird bei der Operatordefinition  
// ein fiktives Argument angegeben.  
cKomplex operator++(int notUsed);
```

Klassen

Überladen von Operatoren

- Überladen von zweistelligen Operatoren (z.B. $a + b$) als Klassenmethode:

```
cKomplex cKomplex::operator+(const cKomplex& obj1) {  
    cKomplex z;  
    z._im = this->_im + obj1._im;  
    z._re = this->_re + obj1._re;  
    return z;  
}
```

- Überladen von zweistelligen Operatoren als “non-member” Funktion:
 - 2 Argumente für jeden Operator
 - Funktion muss als friend der Klasse cKomplex definiert werden

```
cKomplex operator+(const cKomplex& obj1, const cKomplex& obj2) {  
    cKomplex z;  
    z._im = obj1._im + obj2._im;  
    z._re = obj1._re + obj2._re;  
    return z;  
}
```


Klassen

Überladen von Operatoren

- Überladen vom ostream Operator (“ << ”):

```
ostream& operator<<(ostream& os, const cKomplex obj) {  
    os << obj._re << ", " << obj._im;  
    return os;  
}
```

- Danach muss dieser als “friend” von cKomplex definiert werden um auf die privaten Daten zugreifen zu dürfen:

```
class cKomplex {  
    double _re; // Membervariable  
    double _im;  
    friend ostream& operator<<(ostream& os, cKomplex obj);  
    ....  
}
```

- [Webseite](#) mit Information zum Operator Überladen

Klassen

Copy und Move Konstruktor

- Copy-Konstrukturen

- sind Konstruktoren die eine Referenz (in der Regel const) auf eine Instanz einer Klasse erwarten

```
cKomplex::cKomplex(const cKomplex& obj);
```

- kopieren die als Argumente übergebene Instanz(en) der Klasse in das instanzierte Objekt
- werden nicht vom Compiler erzeugt, wenn die Klasse einen Move-Konstruktor oder Move- Zuweisungsoperator besitzt

```
cKomplex operator=(const cKomplex& obj);
```

- Der Compiler benötigt den Copy-Konstruktor, um automatisch Objekte zu kopieren oder neue Objekte aus bestehenden zu initialisieren → wird automatisch von Compiler generiert wenn nicht manuell erstellt

Klassen

Copy und Move Konstruktor

- Move-Konstruktoren
 - sind Konstruktoren, die eine nicht konstante Rvalue-Referenz auf eine Instanz einer Klasse erwarten.

```
cKomplex(cKomplex&& obj);
```
 - besitzen ein ähnliches Einsatzgebiet wie Copy-Konstruktoren.
- verschieben das ursprüngliche Objekt in das zu instanziiierende Objekt.
- Der Move-Konstruktor wird implizit vom Compiler angewandt, um Objekte zu verschieben, die nicht mehr benötigt werden. → Vermeiden von unnötigem Kopieren

Klassen

Copy und Move Konstruktor

- Strategie des Move-Konstruktors
 - 1. Setze die Attribute des neuen Objekts.
 - 2. Verschiebe den Inhalt des alten Objekts in das neue.
 - 3. Setze die Attribute des alten Objekts auf ihre Default-Werte.
- Der Move-Konstruktor wird
 - automatisch erzeugt, wenn alle Attribute einer Klasse und ihrer Basisklassen einen besitzen.
 - nicht erzeugt, wenn für die Klasse ein Copy-Konstruktor, Copy-Zuweisungsoperator, ein Move- Zuweisungsoperator oder ein Destruktor existiert.
- Besitzt eine Klasse einen Move-Konstruktor und einen Copy-Konstruktor, besitzt der Move- Konstruktor Vorrang.

Klassen

Copy und Move Konstruktor

- Besitzt eine Klasse einen Copy-Konstruktor, sollte sie auch einen Copy-Zuweisungsoperator anbieten. Entsprechendes gilt für den Move-Konstruktor und Move-Zuweisungsoperator.
- Eine Klasse unterstützt die Copy-Semantik, wenn sie einen Copy-Konstruktor und einen Copy-Zuweisungsoperator anbietet.
- Eine Klasse unterstützt die Move-Semantik, wenn sie einen Move-Konstruktor und einen Move-Zuweisungsoperator anbietet.
- Die Container der Standard Template Library und der `std::string` bieten Copy- und Move-Semantik an.
- Weiteres unter folgendem [Link](#)

Klassen

Statische Variablen und Methoden

- Statische Membervariablen existieren genau einmal (unabhängig von der Anzahl der Instanzen der Klasse)
- Sind in allen Instanzen der Klasse gleich
- Auch verfügbar, wenn keine Instanz der Klasse existiert
- Statische Variablen müssen initialisiert werden
- Statische Methoden können nur auf statische Variablen Ihrer Klasse zugreifen

```
class Zaehler {  
    static int n;  
public:  
    Zaehler() {n++;}  
    ~Zaehler() {n--;}  
    static int get_n() {return n;}  
};
```

Klassen

Dynamische Speicherallokation

- Der *new* Operator
 - Reserviert Speicherplatz für eine Instanz auf dem Heap (=dynamischer Speicher)
 - Führt den Konstruktor aus und reserviert Speicher beim OS. Nicht mehr benötigter Speicher muss vom Programmierer wieder freigegeben werden. —> **Vorsicht:**
„**Memory-Leak**“
 - Als „Memory-Leak“ oder auch Speicherleiche wird das unnötige Vorhandensein von nicht mehr benötigtem Speicher bezeichnet
 - Liefert den Zeiger auf die Instanz, dieser muss in einer Variable gespeichert werden

```
Zaehler *z = new Zaehler;  
Zaehler *z2 = new Zaehler;  
  
Zaehler *z_ar = new Zaehler[10];
```

Klassen

Dynamische Speicherallokierung

- Der *delete* Operator
 - Analog zu new ruft delete zuerst den Destruktor und gibt reservierten Speicher wieder frei
 - delete erwartet zum Löschen den von new zurückgelieferten Pointer
 - Eine Speicheradresse darf niemals zweimal mit delete gelöscht werden —> Führt zum Programmabsturz. Zur Sicherheit kann man den Pointer auf 0 setzen.

```
delete z;  
delete[] z_ar;
```


Inhalt

- Vorstellung
- Einleitung, Geschichte
- Grundlagen
- Klassen
- **Vererbung**
- Zusätzliche Themen
- Die Standardbibliothek