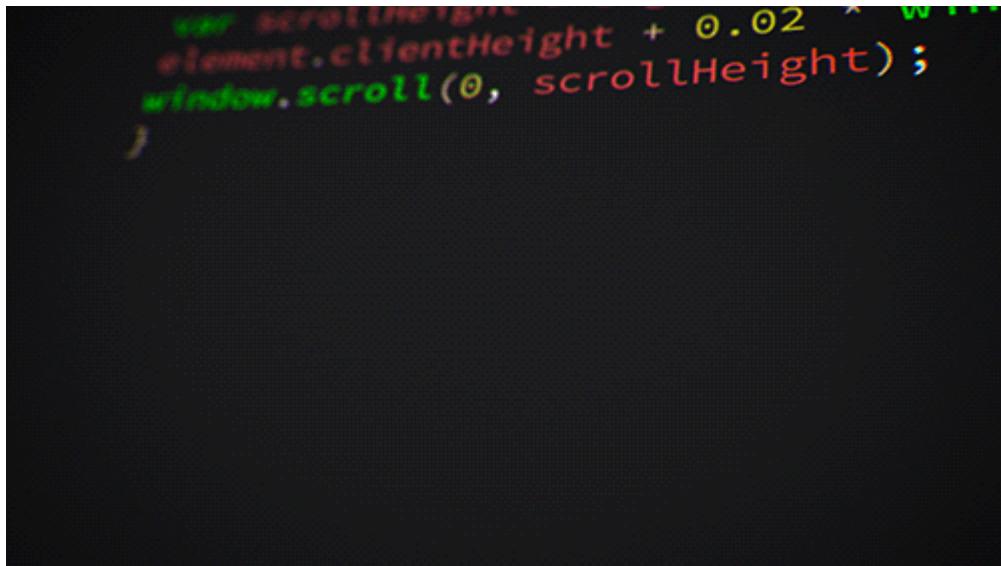


Objektorientierte Programmierung mit C++

Parameter	Kursinformationen
Veranstaltung:	<u>Prozedurale Programmierung / Einführung in die Informatik / Erhebung, Analyse und Visualisierung digitaler Daten</u>
Semester	<u>Wintersemester 2025/26</u>
Hochschule:	<u>Technische Universität Freiberg</u>
Inhalte:	<u>Operatorenüberladung / Vererbung</u>
Link auf Repository:	<u>https://github.com/TUBAF-IfI-LiaScript/VL_EAVD/blob/master/04_Funktionen.md</u>
Autoren	Sebastian Zug & André Dietrich & Galina Rudolf



```
scrHeight = element.clientHeight + 0.02 * w;
window.scroll(0, scrollHeight);
```

Fragen an die heutige Veranstaltung ...

- Was sind Operatoren?
- Warum werden eigene Operatoren für individuelle Klassen benötigt?
- Wann spricht man von Vererbung und warum wird sie angewendet?
- Welche Zugriffsattribute kennen Sie im Zusammenhang mit der Vererbung?

Reflexion Ihrer Fragen / Rückmeldungen

Zur Erinnerung ... Wettstreit zur partizipativen Materialentwicklung mit den Informatikern ...



Preis für das aktivste Auditorium

Format	Informatik Studierende
Verbesserungsvorschlag	3
Fragen	2
generelle Hinweise	0

Die Zustimmung (in Prozent) für die Verwendung der künstlichen Intelligenz im Pflegebereich unter der Bevölkerung von Mauritius und Réunion soll vergleichend betrachtet werden. Die Ergebnisse der Umfragen für die Jahre 2010 bis 2020 (je ein Wert pro Jahr) sollen in zwei Arrays erfasst werden (je ein Array pro Insel) und in einem Programm ausgewertet werden.

- Für beide Inseln soll aus den in Arrays erfassten Werten je ein Mittelwert berechnet werden. Schreiben Sie dazu eine Funktion, die ein Array übergeben bekommt und einen Mittelwert als ein Ergebnis an die main-Funktion zurück liefert. Rufen Sie die Funktion in der main-Funktion für jedes beider Arrays auf und geben Sie die Mittelwerte in der main-Funktion aus.
- Schreiben Sie eine weitere Funktion, die die korrespondierenden Werte beider Arrays miteinander vergleicht. Geben Sie für jedes Jahr aus, auf welcher Insel die Zustimmung größer war, bei den gleichen Werte ist eine entsprechende Meldung auszugeben. Rufen Sie die Funktion in der main-Funktion auf.
- In der main()-Funktion sind die Werte von der Console einzulesen und in die Arrays zu speichern.

Lösung Beispielaufgabe aus der Klausur

example.cpp



```
1 #include <iostream>
2
3 const int YEARS = 11; // von 2010 bis 2020 inkl.
4
5 double berechneMittelwert(const double arr[], int size) {
6     double summe = 0.0;
7     for (int i = 0; i < size; ++i) {
8         summe += arr[i];
9     }
10    return summe / size;
11 }
12
13 void vergleicheZustimmung(const double arrMauritius[], const double
14                             arrReunion[], int size) {
15     std::cout << "\nVergleich der Zustimmung pro Jahr:\n";
16     for (int i = 0; i < size; ++i) {
17         int jahr = 2010 + i;
18         std::cout << "Jahr " << jahr << ":" ;
19         if (arrMauritius[i] > arrReunion[i]) {
20             std::cout << "Mauritius hat höhere Zustimmung (" <<
21                         arrMauritius[i] << "% vs " << arrReunion[i] << "%)\n";
22         } else if (arrMauritius[i] < arrReunion[i]) {
23             std::cout << "Réunion hat höhere Zustimmung (" << arrReun
24                         << "% vs " << arrMauritius[i] << "%)\n";
25         } else {
26             std::cout << "Beide Inseln haben gleiche Zustimmung (" <<
27                         arrMauritius[i] << "%)\n";
28         }
29     }
30 }
31
32 int main() {
33     //Das wäre die korrekte Lösung - um die Eingaben zu vermeiden hab
34     //feste Werte eingetragen
35     //double zustimmungMauritius[YEARS];
36     //double zustimmungReunion[YEARS];
37     //
38     //std::cout << "Geben Sie die Zustimmung in Prozent für Mauritius
39     //die Jahre 2010 bis 2020 ein:\n";
40     //for (int i = 0; i < YEARS; ++i) {
41     //    std::cout << "Jahr " << 2010 + i << ":" ;
42     //    std::cin >> zustimmungMauritius[i];
43     //}
44     //
45     //std::cout << "\nGeben Sie die Zustimmung in Prozent für Réunion
46     //die Jahre 2010 bis 2020 ein:\n";
47     //for (int i = 0; i < YEARS; ++i) {
```

```

42     //     std::cout << "Jahr " << 2010 + i << ":" ;
43     //     std::cin >> zustimmungReunion[i];
44 }
45
46 // Feste Werte für Mauritius
47 double zustimmungMauritius[YEARS] = {65.5, 67.2, 70.1, 72.3, 74.0
48 , 77.8, 80.0, 82.1, 84.3, 85.0};
49 // Feste Werte für Réunion
50 double zustimmungReunion[YEARS] = {60.0, 62.5, 65.0, 68.0, 70.5,
51 75.0, 78.0, 80.0, 83.0, 85.0};
52
53 double mittelwertMauritius = berechneMittelwert(zustimmungMauritius,
54 YEARS);
55 double mittelwertReunion = berechneMittelwert(zustimmungReunion,
56 );
57
58 std::cout << "\nMittelwert der Zustimmung (2010-2020):\n";
59 std::cout << "Mauritius: " << mittelwertMauritius << "%\n";
60 std::cout << "Réunion: " << mittelwertReunion << "%\n";
61
62 vergleicheZustimmung(zustimmungMauritius, zustimmungReunion, YEARS);
63
64 return 0;
65 }
```

Mittelwert der Zustimmung (2010-2020):

Mauritius: 75.8%

Réunion: 72.7273%

Vergleich der Zustimmung pro Jahr:

Jahr 2010: Mauritius hat höhere Zustimmung (65.5% vs 60%)

Jahr 2011: Mauritius hat höhere Zustimmung (67.2% vs 62.5%)

Jahr 2012: Mauritius hat höhere Zustimmung (70.1% vs 65%)

Jahr 2013: Mauritius hat höhere Zustimmung (72.3% vs 68%)

Jahr 2014: Mauritius hat höhere Zustimmung (74% vs 70.5%)

Jahr 2015: Mauritius hat höhere Zustimmung (75.5% vs 73%)

Jahr 2016: Mauritius hat höhere Zustimmung (77.8% vs 75%)

Jahr 2017: Mauritius hat höhere Zustimmung (80% vs 78%)

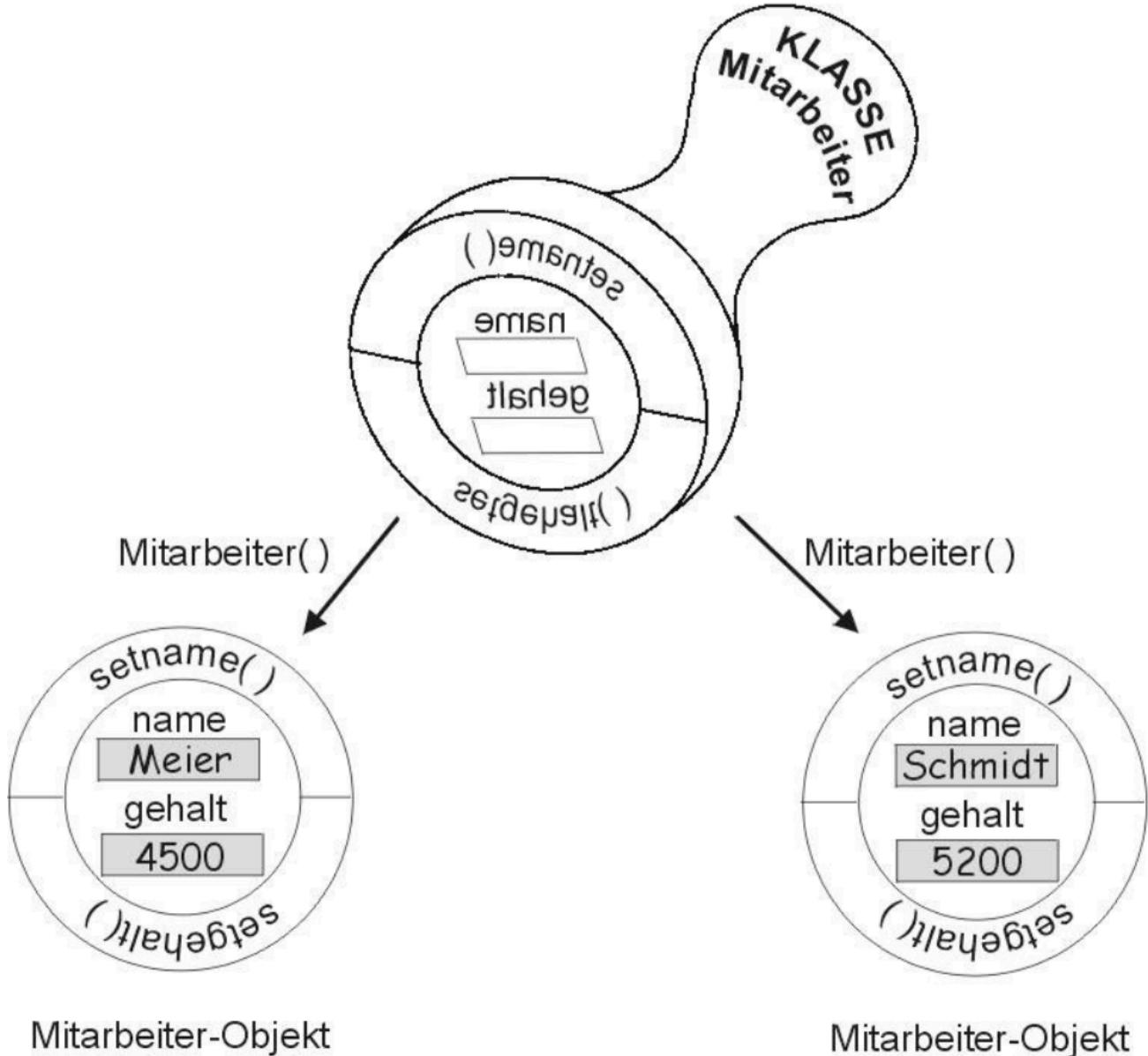
Jahr 2018: Mauritius hat höhere Zustimmung (82.1% vs 80%)

Jahr 2019: Mauritius hat höhere Zustimmung (84.3% vs 83%)

Jahr 2020: Beide Inseln haben gleiche Zustimmung (85%)

Rückblick

Unter einer Klasse (auch Objekttyp genannt) versteht man in der objektorientierten Programmierung ein abstraktes Modell bzw. einen Bauplan für eine Reihe von ähnlichen Objekten.



Von Binz - Own Creation, CC BY-SA 4.0, <https://commons.wikimedia.org/w/index.php?curid=62707688>

Operatorenüberladung

Motivation

Folgendes Beispiel illustriert den erreichten Status unserer C++ Implementierungen. Unsere Klasse **Student** besteht aus:

- 3 Membervariablen (Zeile 5-7)
- 2 Konstruktoren (Zeile 9-10)
- 1 Memberfunktion (Zeile 12)

Alle sind als `public` markiert.

example.cpp



```

1 #include <iostream>
2
3 class Student {
4     public:
5         std::string name;
6         int alter;
7         std::string ort;
8
9     Student(std::string n);
10    Student(std::string n, int a, std::string o);
11
12    void ausgabeMethode(std::ostream& os); // Deklaration der Methode
13 };
14
15 Student::Student(std::string n):
16     name(n), alter(8), ort("Freiberg")
17 {}
18
19 Student::Student(std::string n, int a, std::string o):
20     name(n), alter(a), ort(o)
21 {}
22
23 void Student::ausgabeMethode(std::ostream& os) {
24     os << name << " " << ort << " " << alter << "\n";
25 }
26
27 int main()
28 {
29     Student gustav = Student("Zeuner", 27, "Chemnitz");
30     //Student gustav {"Zeuner", 27, "Chemnitz"};
31     //Student gustav("Zeuner", 27, "Chemnitz");
32     gustav.ausgabeMethode(std::cout);
33
34     Student bernhard {"Cotta", 18, "Zillbach"};
35     bernhard.ausgabeMethode(std::cout);
36
37     Student nochmalBernhard {"Cotta", 18, "Zillbach"};
38 }
```

Zeuner Chemnitz 27
Cotta Zillbach 18
Zeuner Chemnitz 27
Cotta Zillbach 18

Aufgabe: Schreiben Sie

- eine Funktion `int vergleich(Student, Student)` und
- eine Methode `int Student::vergleich(Student)`,

die zwei Studenten miteinander vergleicht!

Was ist der konzeptionelle Unterschied zwischen beiden Implementierungen?

Frage: Was ist der Nachteil unserer Lösung?

Konzept

Das Überladen von Operatoren erlaubt die flexible klassenspezifische Nutzung von Arithmetischen- und Vergleichs-Symbolen wie `+`, `-`, `*`, `==`. Damit kann deren Bedeutung für selbstdefinierte Klassen mit einer neuen Bedeutung versehen werden. Ausnahmen bilden spezielle Operatoren, die nicht überladen werden dürfen (`?:`, `::`, `..`, `.*`, `typeid`, `sizeof` und die Cast-Operatoren).

```
Matrix a, b;
Matrix c = a + b;      \\ hier wird mit dem Plus eine Matrixoperation ausgeführt

String a, b;
String c = a + b;      \\ hier werden mit dem Plus zwei Strings konateniert
```

Operatorüberladung ist Funktionsüberladung, wobei die Funktionen durch eine spezielle Namensgebung gekennzeichnet sind. Diese beginnen mit dem Schlüsselwort `operator`, das vom Symbol für den jeweiligen Operator gefolgt wird.

```
class Matrix {
public:
    Matrix operator+(Matrix zweiterOperand) { ... }
    Matrix operator/(Matrix zweiterOperand) { ... }
    Matrix operator*(Matrix zweiterOperand) { ... }
}

class String {
public:
```

```
String operator+(String zweiterString) { ... }  
}
```

Operatoren können entweder als Methoden der Klasse oder als globale Funktionen überladen werden. Die Methodenbeispiele sind zuvor dargestellt, analoge Funktionen ergeben sich zu:

```
class Matrix {  
    public:  
        ...  
}  
  
Matrix operator+(Matrix ersterOperand, Matrix zweiterOperand) { ... }  
Matrix operator/(Matrix ersterOperand, Matrix zweiterOperand) { ... }  
Matrix operator*(Matrix ersterOperand, Matrix zweiterOperand) { ... }
```

Merke: Funktion oder Methode - welche Version sollte wann zum Einsatz kommen? Einstellige Operatoren **++** sollten Sie als Methode, zweistellige Operatoren ohne Manipulation der Operanden als Funktion implementieren. Für zweistellige Operatoren, die einen der Operanden verändern (**+ =**), sollte als Methode realisiert werden.

Als Beispiel betrachten wir eine Klasse Rechteck und implementieren zwei Operatorüberladungen:

- eine Vergleichsoperation
- eine Additionsoperation die **A = A + B** oder abgekürzt **A+=B**

implementiert.

Comparison.cpp



```
1 #include <iostream>
2
3 class Rectangle {
4     private:
5         float width, height;
6     public:
7         Rectangle(int w, int h):
8             width{w}, height{h}
9         {}
10        float area() {
11            return width * height;
12        }
13        Rectangle operator+=(Rectangle offset) {
14            float ratio = (offset.area() + this->area()) / this->area();
15            this->width = ratio * this->width;
16            return *this;
17        }
18    };
19
20        bool operator>(Rectangle a, Rectangle b) {
21            if (a.area() > b.area())
22                return 1;
23            else
24                return 0;
25        }
26
27 int main () {
28     Rectangle rect_a(3,4);
29     Rectangle rect_b(5,7);
30     std::cout << "Vergleich: " << (rect_a > rect_b) << "\n";
31
32     std::cout << "Fläche a : " << rect_a.area() << "\n";
33     std::cout << "Fläche b : " << rect_b.area() << "\n";
34     rect_a += rect_b;
35     std::cout << "Summe      : " << rect_a.area();
36
37     return 0;
38 }
```

```
Vergleich: 0
Fläche a : 12
Fläche b : 35
Summe     : 47Vergleich: 0
Fläche a : 12
Fläche b : 35
Summe     : 47
```

Merke: Üblicherweise werden die Operanden, welche lediglich betrachtet werden, bei der Operatorüberladung als Referenzen übergeben. Damit wird eine Kopie vermieden. In Kombination mit dem Schlüsselwort `const` kann dem Compiler angezeigt werden, dass keine Veränderung an den Daten vorgenommen wird. Sie müssen also nicht gespeichert werden.

<!-- TODO: Aufgabe: Überlegen Sie sich, in welchen Situationen ein pass by reference eventuell doch sinnvoll wäre. -->

```
bool operator>(const Rectangle& a, const Rectangle& b) {
    if (a.area() > b.area())
        return 1;
    else
        return 0;
}
```

Stellen wir die Abläufe nochmals grafisch dar [Pythontutor](#)

Anwendung

Im folgenden Beispiel wird der Vergleichsoperator `==` überladen. Dabei sehen wir den Abgleich des Namens und des Alters als ausreichend an.

Comparison.cpp



```
1 #include <iostream>
2
3 class Student {
4     public:
5         std::string name;
6         int alter;
7         std::string ort;
8
9         Student(std::string n);
10        Student(std::string n, int a, std::string o);
11
12        void ausgabeMethode(std::ostream& os); // Deklaration der Methode
13        bool operator==(const Student&);
14    };
15
16 Student::Student(std::string n):
17     name(n), alter(8), ort("Freiberg")
18 {}
19
20 Student::Student(std::string n, int a, std::string o):
21     name(n), alter(a), ort(o)
22 {}
23
24 void Student::ausgabeMethode(std::ostream& os) {
25     os << name << " " << ort << " " << alter << "\n";
26 }
27
28 bool Student::operator==(const Student& other) {
29     if ((this->name == other.name) && (this->alter == other.alter))
30         return true;
31     else
32         return false;
33 }
34
35 int main()
36 {
37     Student gustav = Student("Zeuner", 27, "Chemnitz");
38     gustav.ausgabeMethode(std::cout);
39
40     Student bernhard {"Cotta", 18, "Zillbach"};
41     bernhard.ausgabeMethode(std::cout);
42
43     Student NochMalBernhard {"Cotta", 18, "Zillbach"};
44     NochMalBernhard.ausgabeMethode(std::cout);
45
46     if (bernhard == NochMalBernhard)
47         std::cout << "Identische Studenten\n";
48     else
```

```
49     std::cout << "Ungleiche Identitäten\n";
50 }
```

```
Zeuner Chemnitz 27
Cotta Zillbach 18
Cotta Zillbach 18
Identische Studenten
Zeuner Chemnitz 27
Cotta Zillbach 18
Cotta Zillbach 18
Identische Studenten
```

Eine besondere Form der Operatorüberladung ist der `<<`, mit dem die Ausgabe auf ein Streamobjekt realisiert werden kann.

Eine umfangreiche Diskussion zur Operatorüberladung finden Sie unter <https://wwwcplusplus.net/forum/topic/232010/%C3%BCberladung-von-operatoren-in-c-teil-1/2>

Ebenso eine ausführliche Auskunft über Operatoren und ihre Eigenheiten:
<https://en.cppreference.com/w/cpp/language/operators.html>

Vererbung

MultipleTypesOfPeople.cpp



```
1 #include <iostream>
2
3 class Student {
4     public:
5         std::string name;
6         std::string ort;
7         std::string studiengang;
8
9         Student(std::string n, std::string o, std::string sg):
10             name{n}, ort{o}, studiengang{sg}
11     {}
12     void printCertificate(std::ostream& os) {
13         os << "Studentendatensatz: " << name << " " << ort << " " <<
14             studiengang << "\n";
15     }
16 };
17 int main() {
18     Student gustav = Student("Zeuner", "Chemnitz", "Mathematik");
19     gustav.printCertificate(std::cout);
20
21 // Professor winkler = Professor("Winkler", "Freiberg");
22 // winkler.printCertificate(std::cout);
23 }
```

Studentendatensatz: Zeuner Chemnitz Mathematik

Studentendatensatz: Zeuner Chemnitz Mathematik

Aufgabe: Implementieren Sie eine neue Klasse **Professor**, die aber auf die Membervariable **Studiengang** verzichtet, aber eine neue Variable **Fakultät** einführt.

Motivation

Merke: Eine unserer Hauptmotivationen bei der "ordentlichen" Entwicklung von Code ist die Vermeidung von Code-Duplikationen!

In unserem Code entstehen Dopplungen, weil bestimmte Variablen oder Memberfunktionen usw. mehrfach für individuelle Klassen implementiert werden. Dies wäre für viele Szenarien analog der Fall:

Basisklasse	abgeleitete Klassen	Gemeinsamkeiten
Fahrzeug	Flugzeug, Boot, Automobil	Position, Geschwindigkeit, Zulassungsnummer, Führerscheinpflicht
Datei	Foto, Textdokument, Datenbankauszug	Dateiname, Dateigröße, Speicherort
Nachricht	Email, SMS, Chatmessage	Adressat, Inhalt, Datum der Versendung

Merke: Die Vererbung ermöglicht die Erstellung neuer Klassen, die ein in existierenden Klassen definiertes Verhalten wieder verwenden, erweitern und ändern. Die Klasse, deren Member vererbt werden, wird Basisklasse genannt, die erbende Klasse als abgeleitete Klasse bezeichnet.

Implementierung in C++

In C++ werden Vererbungsmechanismen folgendermaßen abgebildet:

```
class Fahrzeug {
public:
    int aktuellePosition[2];      // lat, long Position auf der Erde
    std::string zulassungsnummer;
    bool fuehrerscheinpflichtig
    ...
};

class Flugzeug: public Fahrzeug {
public:
    int flughoehe;
    void fliegen();
    ...
};

class Boot: public Fahrzeug {
public:
    void schwimmen();
    ...
};
```

Die allgemeinere Klasse **Fahrzeug** liefert einen Bauplan für die spezielleren Klassen, die diese Vorgaben weiter ergänzen. Folglich müssen wir uns die Frage stellen, welche Daten oder Funktionalität übergreifend abgebildet werden soll und welche individuell realisiert werden sollen.

Dabei können ganze Ketten von Vererbungen entstehen, wenn aus einem sehr allgemeinen Objekt über mehrere Stufen eine spezifischere Zusammenstellung von Membern umgesetzt wird.

```
class Fahrzeug {  
public:  
    int aktuellePosition[2];      // lat, long Position auf der Erde  
    std::string zulassungsnummer;  
    bool fuehrerscheinpflichtig  
    ...  
};  
  
class Automobil: public Fahrzeug {  
public:  
    void fahren();  
    int zahlderRaeder;  
    int sitze;  
    ...  
};  
  
class Hybrid: public Automobil {  
public:  
    void fahreElektrisch();  
    ...  
};
```

Was bedeutet das für unsere Implementierung von Studenten und Professoren?

Inheritance.cpp



```
1 #include <iostream>
2
3 class Student {
4     public:
5         std::string name;
6         std::string ort;
7         std::string studiengang;
8
9         Student(std::string n, std::string o, std::string sg):
10             name{n}, ort{o}, studiengang{sg}
11     {};
12     void printCertificate(std::ostream& os) {
13         os << "Studentendatensatz: " << name << " " << ort << " " <<
14             studiengang << "\n";
15     }
16 };
17 int main()
18 {
19     Student gustav = Student("Zeuner", "Chemnitz", "Mathematik");
20     gustav.printCertificate(std::cout);
21
22     //Professor winkler = Professor("Winkler", "Freiberg");
23     //winkler.printCertificate(std::cout);
24 }
```

```
Studentendatensatz: Zeuner Chemnitz Mathematik
```

```
Studentendatensatz: Zeuner Chemnitz Mathematik
```

Ein weiteres Beispiel greift den Klassiker der Einführung objektorientierter Programmierung auf, den Kanon der Haustiere 😊 Das Beispiel zeigt die Initialisierung der Membervariablen :

- der Basisklasse beim Aufruf des Konstruktors der erbenden Klasse
- der Member der erbenden Klasse wie gewohnt

Animals.cpp



```
1 #include <iostream>
2
3 // Vererbende Klasse Animal
4 class Animal {
5 public:
6     Animal():
7         name{"Animal"}, weight{0.0}
8     {}
9     Animal(std::string _name, double _weight):
10        name{_name}, weight{_weight}
11    {}
12     void sleep () {
13         std::cout << name << " is sleeping!\n";
14     }
15
16     std::string name;
17     double weight;
18 };
19
20 // Erbende Klasse Dog – Dog übernimmt die Methoden und Attribute von
21 class Dog: public Animal {
22 public:
23     Dog(std::string name, double weight, int id):
24         Animal(name, weight), id{id}
25     {}
26
27     // Dog-spezifische Methoden: bark() und top_speed()
28     void bark() {
29         std::cout << "woof woof\n";
30     }
31     double top_speed() {
32         if (weight < 40) return 15.5;
33         else if (weight < 90) return 17.0;
34         else return 16.2;
35     }
36
37     int id;
38 };
39
40 int main() {
41     Dog dog = Dog("Rufus", 50.0, 2342);
42     dog.sleep();
43     dog.bark();
44     std::cout << dog.top_speed() << "\n";
45 }
```

```
Rufus is sleeping!
```

```
woof woof
```

```
17
```

```
Rufus is sleeping!
```

```
woof woof
```

```
17
```

Überschreiben von Methoden der Basisklasse (Polymorphie)

Die grundsätzliche Idee bezieht sich auf die Implementierung "eigener" Methoden gleicher Signatur in den abgeleiteten Klassen. Diese implementieren dann das spezifische Verhalten der jeweiligen Objekte.

Inheritance.cpp



```
1 #include <iostream>
2
3 class Person {
4     public:
5         std::string name;
6         std::string ort;
7
8         Person(std::string n, std::string o):
9             name{n}, ort{o}
10    {}
11    void printData(std::ostream& os) {
12        os << "Datensatz: " << name << " " << ort << "\n";
13    }
14 };
15
16 class Student: public Person {
17     public:
18         std::string studiengang;
19
20         Student(std::string n, std::string o, std::string sg):
21             Person(n, o), studiengang{sg}
22         {}
23         void printData(std::ostream& os) {
24             os << "Student Datensatz: " << name << " " << ort << "\n";
25         }
26     };
27
28 class Professor: public Person {
29     public:
30         int id;
31
32         Professor(std::string n, std::string o, int id):
33             Person(n, o), id{id}
34         {}
35         void printData(std::ostream& os) {
36             os << "Prof. Datensatz: " << name << " " << ort << "\n";
37         }
38     };
39
40 int main()
41 {
42     Student *gustav = new Student("Zeuner", "Chemnitz", "Mathematik");
43     gustav->printData(std::cout);
44     delete gustav;
45
46     Professor *winkler = new Professor("Winkler", "Freiberg", 234234);
47     winkler->printData(std::cout);
48     delete winkler;
```

Student Datensatz: Zeuner Chemnitz

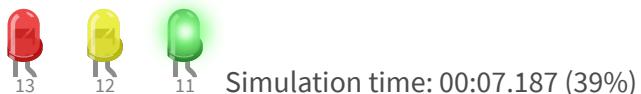
Prof. Datensatz: Winkler Freiberg

Student Datensatz: Zeuner Chemnitz

Prof. Datensatz: Winkler Freiberg

Anwendungsbeispiele

Entwerfen Sie eine Klasse, die das Verhalten einer Ampel mit den notwendigen Zuständen modelliert. Welche Methoden sollten zusätzlich in die Klasse aufgenommen werden?



TrafficLight.cpp



```
1 class Ampel {
2     private:
3         int redPin, yellowPin, greenPin;
4         int state = 0;
5
6     public:
7         Ampel(int red, int yellow, int green):
8             redPin{red}, yellowPin{yellow}, greenPin{green}
9         {
10             pinMode(red, OUTPUT);
11             pinMode(yellow, OUTPUT);
12             pinMode(green, OUTPUT);
13         };
14         void activateRed() {
15             digitalWrite(redPin, HIGH);
16         }
17         void startOnePeriod(int waitms) {
18             digitalWrite(redPin, HIGH);
19             delay(waitms);
20             digitalWrite(yellowPin, HIGH);
21             delay(waitms);
22             digitalWrite(redPin, LOW);
23             digitalWrite(yellowPin, LOW);
24             digitalWrite(greenPin, HIGH);
25         }
26     };
27
28     void setup() {
29         Ampel trafficLight = Ampel(13, 12, 11);
30         trafficLight.activateRed();
31         trafficLight.startOnePeriod(1000);
32     }
33
34     void loop() {
35         delay(100);
36     }
}
```

Sketch uses 1016 bytes (3%) of program storage space. Maximum is 32256 bytes.

Global variables use 9 bytes (0%) of dynamic memory, leaving 2039 bytes for local variables. Maximum is 2048 bytes.

Sketch uses 1016 bytes (3%) of program storage space. Maximum is 32256 bytes.

Global variables use 9 bytes (0%) of dynamic memory, leaving 2039 bytes for local variables. Maximum is 2048 bytes.

Allgemeine C++ Anwendungsbeispiele

Nachfolgend sollen einige Anwendungsbeispiele die Möglichkeiten von C++ in Bezug auf Ihre Fachdisziplinen aufzeigen.

- Messung der Ausbreitung von Schwingungen in Festkörpern
- Bildanalyse mit OpenCV am Beispiel von Dünnschliffen

Beide Beispiele finden Sie im Repository unter [Link](#)

Quiz

Operatorenüberladung

Konzept

Welches Schlüsselwort wird bei der Operatorüberladung verwendet?

Für welche Operatoren sollten die Methoden einer Klasse und für welche die globalen Funktionen bevorzugt zum Einsatz kommen?

Methode	Funktion	
<input type="radio"/>	<input type="radio"/>	<input type="checkbox"/> ++
<input type="radio"/>	<input type="radio"/>	<input type="checkbox"/> +
<input type="radio"/>	<input type="radio"/>	<input type="checkbox"/> *
<input type="radio"/>	<input type="radio"/>	<input type="checkbox"/> %
<input type="radio"/>	<input type="radio"/>	<input type="checkbox"/> --
<input type="radio"/>	<input type="radio"/>	<input type="checkbox"/> +=

Wie lautet die Ausgabe dieses Programms? (Hinweise können über das Feld mit der Glühbirne angezeigt werden.)

```
#include<iostream>

class Vektor {
public:
    int x = 0, y = 0;

    Vektor(int x, int y);

    Vektor operator+(const Vektor& vek_tmp);

    void printVektor() {
        std::cout << x << ", " << y << "\n";
    }
};

Vektor::Vektor(int x, int y):
    x(x), y(y)
{};

Vektor Vektor::operator+(const Vektor& vek_tmp) {
    Vektor vektor;
    vektor.x = x + vek_tmp.x;
    vektor.y = y + vek_tmp.y;
    return vektor;
}
```

```

    vektor vek_loe(0, 0);
    vek_loe.x = this->x + vek_tmp.x;
    vek_loe.y = this->y + vek_tmp.y;
    return vek_loe;
}

int main() {
    Vektor v1(4, 7), v2(10, 9);
    Vektor v3 = v1 + v2;
    v3.printVektor();
}

```

Anwendung

Wie lautet die Ausgabe dieses Programms?

```

#include <iostream> □

class Ort {
public:
    std::string name;
    std::string bundesland;
    int einwohner;

    Ort(const Ort&); ...
    Ort(std::string n); ...
    Ort(std::string n, std::string b, int e); ...
};

Ort::Ort(std::string n, std::string b, int e):
    name{n}, bundesland{b}, einwohner{e}
{ }

std::ostream& operator<<(std::ostream& os, const Ort& ort)
{
    os << ort.name << ", " << ort.bundesland << ", " << ort.einwohner;
    return os;
}

int main()
{
    Ort Freiberg = Ort("Freiberg", "Sachsen" , 41823);
    std::cout << Freiberg << "\n";
}

```

Vererbung

Erbt die erbende Klasse immer alle Attribute und Methoden der Basisklasse?

- Ja
- Nein

Implementierung in C++

Wodurch muss [_____] ersetzt werden um eine Klasse **Flugzeug** zu definieren, die von der Klasse **Fahrzeug** erbt?

```
#include <iostream>

class Fahrzeug {
public:
    int aktuellePosition[2];
    std::string Zulassungsnummer;
    bool Fuehrerscheinpflichtig;
};

[_____]{
public:
    int Flughoehe;
    void fliegen();
};
```



Kann eine abgeleitete Klasse als Basis-Klasse für eine weitere Klasse verwendet werden?

Ja Nein

Wie lautet die Ausgabe dieses Programms?

```
#include <iostream>

class Fahrzeug {
public:
    Fahrzeug():
        name{"Fahrzeug"}
    {}
    Fahrzeug(std::string _name):
        name{_name}
    {}
    void defekt() {
        std::cout << name << " muss in die Werkstatt.\n";
    }

    std::string name;
};

class Auto: public Fahrzeug {
public:
    Auto(std::string name, int ps):
        Fahrzeug(name), ps{ps}
    {}
    int ps = 0;
};

int main() {
    Auto auto1 = Auto("Peters Auto", 100);
    auto1.defekt();
    return 0;
}
```

Vererbungsattribute

Welche Zugriffspezifizierer sind für die Mitglieder einer Basisklasse zu verwenden damit die folgenden Aussagen zutreffen?

private	protected	public	
<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Zugriff ist aus einer beliebigen Klasse möglich.
<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Zugriff ist nur innerhalb der Basisklasse möglich.
<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	Zugriff ist in der Basisklasse und in erbenden Klassen möglich.

Polymorphie

Was ist Polymorphie?

- Eine Technik, die es verhindert, bestehende Methoden in den ableiteten Klassen aufzurufen
- Eine Technik, die es ermöglicht, bestehende Methoden zu überschreiben
- Eine Technik, die es ermöglicht, Datenfelder einer Klasse in den abgeleiteten Klassen zu überschreiben

Welche Aussagen treffen im Bezug auf Polymorphie zu?

- Polymorphie soll beim Erstellen einer abgeleiteten Klasse immer durch Vergabe eines anderen Namens umgangen werden.
- Polymorphie ermöglicht die Methoden der Basisklasse in der abgeleiteten Klasse mit einer anderen Funktionalität zu versehen.
- Polymorphie wird verwendet um die Methoden der Basisklasse zu löschen.

Wie lautet die Ausgabe dieses Programms?

```
#include <iostream>

class Basisklasse {
public:
    void ausgabe() {
        std::cout << "Ausgabe1\n";
    }
}
```



```
};

class Ableitungsklasse : public Basisklasse {
public:
    void ausgabe() {
        std::cout << "Ausgabe2\n";
    }
};

int main() {
    Basisklasse b = Basisklasse();
    b.ausgabe();

    Ableitungsklasse a = Ableitungsklasse();
    a.ausgabe();
    return 0;
}
```