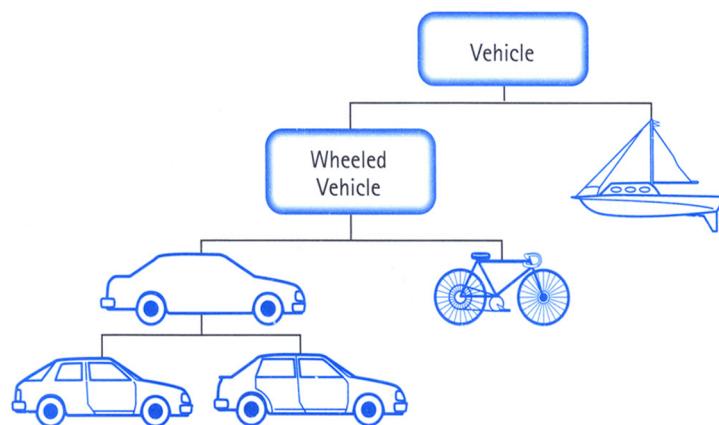




Objekt-Orientierte Programmierung

VORLESUNG 4





Overview

- Wiederholungen
 - Syntax
 - Konstruktoren
- OOP
 - Vererbung
 - Substitutionsprinzip
- UML Diagramme



Syntax Wiederholung

Expression	can be read as
x	variable for a value / object
*x	pointed by x
&x	address of x
x.y	member y of object x
x->y	member y of object pointed by x
(*x).y	member y of object pointed by x (equivalent to the previous one)
x[0]	first object pointed by x
x[1]	second object pointed by x
x[n]	(n+1)th object pointed by x



Beispiel – einfache Klasse

vorlesung4_Punkt.cpp

```
class Punkt
{
private:
    short x;
    short y;

public:
    Punkt(int cx, int cy)
    {
        x = cx;
        y = cy;
    }
    short const get_x() { return x; }
    short const get_y() { return y; }
    void set(short sx, short sy)
    {
        x = sx;
        y = sy;
    }
};
```

```
int main()
{
    float x = 1.0;
    Punkt P = Punkt(0, 0);
    P.set((int)x, 5);
    P.set(P.get_x(), 123);
    return 0;
}
```



Statische Elemente I

Statische Attribute (Klassenattribute)

- statische Attribute gehören zur Klasse
- sie repräsentieren keinen Objektzustand
- sie sind "global" für alle Objekte der Klasse. D.h. sie werden von allen Objekten geteilt
- Zugriff auf solche Variable erfolgt mit den **Klassenname** und dem **Scope-Operator** (::)



Statische Elemente II

Statische Methoden (Klassenmethoden)

- gehören zur Klasse
- keine vorhanden Instanz der Klasse ist erforderlich
- können nur auf andere statische **Attribute/Methoden** bzw. Funktionen außerhalb der Klasse zugreifen
- haben keinen Zugriff auf den **this** Zeiger
- Zugriff auf solche Methoden erfolgt mit den **Klassenname** und dem **Scope-Operator** (`::`)



Beispiel – Statische Elemente

`vorlesung4_Box.cpp`

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

class Box
{
public:
    // Constructor definition
    Box(double l = 2.0, double b = 2.0, double h = 2.0)
    {
        cout << "Constructor called." << endl;
        length = l;
        breadth = b;
        height = h;

        // Increase every time object is created
        objectCount++;
    }
    double Volume()
    {
        return length * breadth * height;
    }
    static int getCount()
    {
        return objectCount;
    }

private:
    static int objectCount;
    double length; // Length of a box
    double breadth; // Breadth of a box
    double height; // Height of a box
};
```

```
// Initialize static member of class Box
int Box::objectCount = 0;

int main(void)
{
    // Print total number of objects before
    // creating object.
    cout << "Initial Box Count: " <<
    Box::getCount() << endl;

    Box Box1(3.3, 1.2, 1.5); // Declare box1
    Box Box2(8.5, 6.0, 2.0); // Declare box2

    // Print total number of objects after
    // creating object.
    cout << "Final Box Count: " <<
    Box::getCount() << endl;

    return 0;
}
```



Konstruktoren und Destruktor

Konstruktoren werden aufgerufen

- wenn eine neue Variable auf dem Stack deklariert wird
- wenn wir die Instanz mit `new` (auf dem Heap) erstellen
- wenn eine Kopie der Instanz erforderlich ist (Kopierkonstruktor):
 - Zuweisung
 - Funktionsargumente Übergabe
 - pass by value
 - Zurückgeben eines Objekts in Funktionen

Der Destruktor wird aufgerufen:

- wenn `delete` verwendet wird, um den Speicher freizugeben
- wenn eine auf dem Stack definierte Instanz den Scope verlässt



Beispiel – Konstruktoren

vorlesung4_Game.cpp

```
#include <iostream>
using namespace std;
class Game
{
private:
    short size;
    char *field;

public:
    short get_size() { return size; }

    char get_field(int pos)
    {
        if (pos >= size)
            return '\0';
        return field[pos];
    }

    Game(short size = 16) // constructor
    {
        cout << "constr " << size << endl;
        this->size = size;
        field = new char[size];
    }

    ~Game()
    {
        if (field != NULL) {
            delete[] field;
            field = NULL;
        }
        cout << "destr" << endl;
    }
}
```



Beispiel – Konstruktoren

```
Game(const Game &game) ;  
  
{  
    cout << "copy ctr" << endl;  
    size = game.size;  
    field = new char[game.size];  
    for (int i = 0; i < size; i++)  
        field[i] = game.field[i];  
}  
  
Game &operator=(const Game &game)  
{  
    cout << "ass oper" << endl;  
    if (this == &game) // self assig.  
        return *this;  
    if (field != NULL)  
        delete[] field;  
    size = game.size;  
    field = new char[game.size];  
    for (int i = 0; i < size; i++)  
        field[i] = game.field[i];  
    return *this;  
}  
  
int main()  
{  
    Game g0; // default constructor  
    Game g1(31); // parameterized  
constructor  
    Game g2(g0); // copy constructor  
    Game g3; // default constructor  
  
    g3 = g0; // assignment operator  
    cout << g3.get_size() << endl;  
  
    return 0;  
}
```



Fragen und Antworten





OOP Grundbegriffe

- **Kapselung:** Gruppierung von Daten und Funktionen als Objekte.
- **Abstraktion:** Klassen für Typen von ähnlichen Objekten. Trennung der Spezifikation eines Objekts von seiner Implementierung
- **Vererbung:** Erlaubt Code zwischen verwandten Typen wiederzuverwenden
- **Polymorphismus:** Ein Objekt kann einer von mehreren Typen sein. Abhängig von seinem Typ wird seinem Verhalten zur Laufzeit bestimmt



Vererbung/Inheritance I

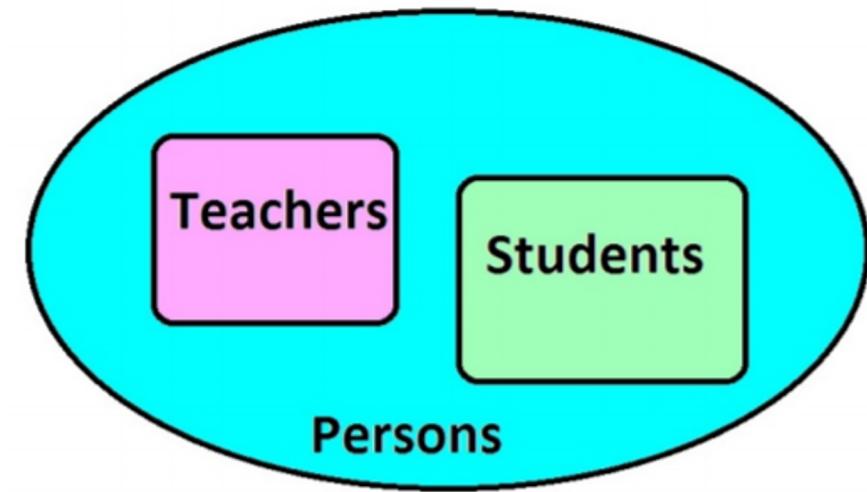
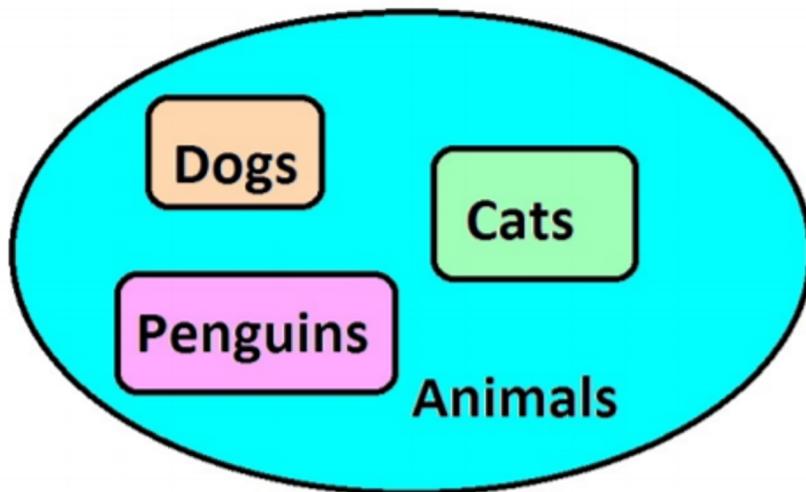
- Ermöglicht ein Definieren einer **neuen Klasse** (Unterklasse) unter Angabe einer **anderen Klasse** (Oberklasse).
- **Vererbung** ermöglicht **Wiederverwendung**.
- Der Aufwand der Entwicklung eines Programms wird reduziert. Software ist robuster.



Vererbung II

- Durch Vererbung können neue Klassen erstellt werden, ohne vorhandene Klasse zu verändern.
- Die neue Klasse hat alle Funktionen der alten Klasse und fügt neue Funktionen hinzu.
- Vererbung kann verwendet werden, wenn es eine **“is a/kind of”-Beziehung** zwischen den Objekten gibt.

Beispiele





Einfache Vererbung I

- mindestens zwei Klassen sind erforderlich: eine **Basisklasse** und eine **abgeleitete Klasse**.
- Wenn **B** und **D** zwei Klassen sind,
 - **D** erbt von **B** oder
 - **D** ist abgeleitet von **B** oder
 - **D** ist eine Spezialisierung von **B**
- das heißt:
 - **D** enthält alle Variablen und Methoden der Klasse **B**;
 - **D** kann Methoden der Klasse **B** umschreiben;
 - **D** kann neue Attribute hinzufügen.



Einfache Vererbung II

- Wenn Klasse D von Klasse B erbt, dann:
 - Ein Objekt der Klasse D enthält alle Attribute der Klasse B;
 - Die Methoden der Klasse B können durch Objekte der Klasse D aufgerufen werden (außer wenn sie versteckt sind).

Syntax

```
class D: public B
{
    // ...
};
```



Einfache Vererbung III

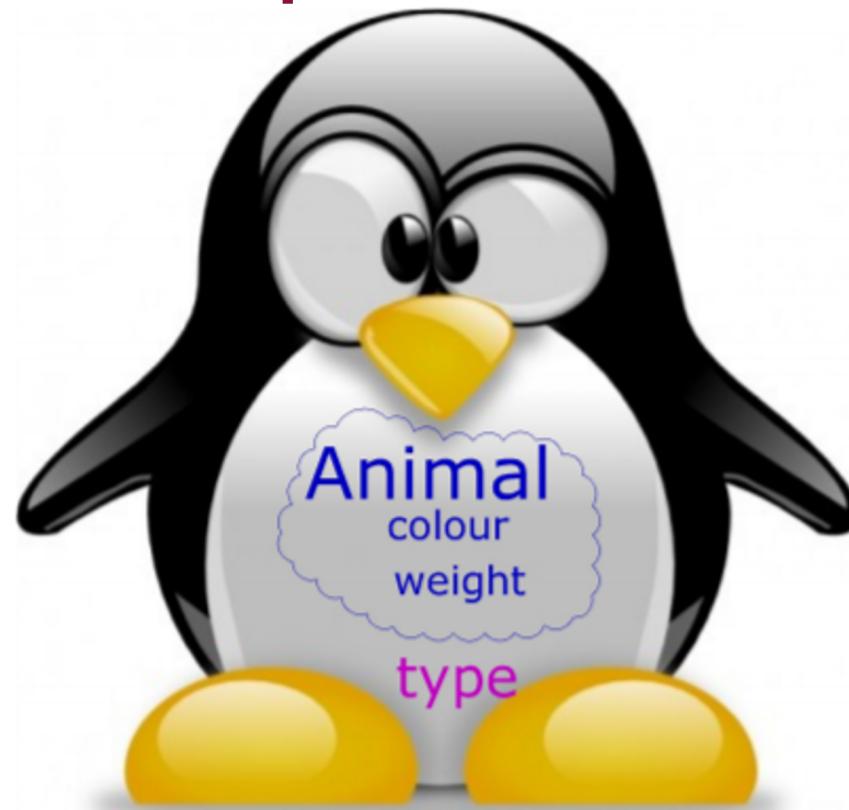
```
class Animal
{
protected:
    std::string colour;
    double weight;
// ...
};

class Penguin: public Animal
{
private:
    std::string type;
// ...
};
```



Einfache Vererbung IV

tier_klassen_demo.zip



github.com/djknoll/oop_tier_klassen_demo.git



Einfache Vererbung V

Fachbegriffe

Syntax

```
class D: public B
{
// ...
};
```

- B = Oberklasse, Basisklasse, Elternklasse.
- D = Unterklasse, abgeleitete Klasse, Subklasse, Kindklasse
- geerbte Elemente (Methoden, Attribute) = definiert in B und unverändert in D
- umdefinierte Elemente (overridden/überschreiben) = definiert in B und D.
- hinzugefügte Element (added) = nur in D definiert



Zugriffsmodifikatoren I

Zugriff auf Members...

- **private:** können von innerhalb der Klasse oder von Freunden Funktionen/Klassen zugegriffen werden
- **protected:** Zugriff ist möglich aus der Klasse selbst und seinen abgeleiteten Klassen, oder Friend Funktionen/Klassen
- **public:** können von überall zugegriffen werden.



Zugriffsmodifikatoren II

Access	public	protected	private
Class	Yes	Yes	Yes
Derived class	Yes	Yes	No
Client code	Yes	No	No



Zugriffsmodifikatoren III

public Vererbung:

Die Zugangsberechtigungen der Attribute der Basisklasse sind nicht geändert.

```
class A: public B { ... }
```

protected Vererbung:

Geerbte public oder protected Attribute der Basisklasse werden zu protected Members in der abgeleiteten Klasse.

```
class A: protected B { ... }
```

private Vererbung:

Geerbte public oder protected Attribute der Basisklasse werden zu private Members in der abgeleiteten Klasse.

```
class A: private B { ... }
```



Zugriffsmodifikatoren IV

Inheritance type	public	protected	private
Base access specifier	Derived access specifier		
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Private	Private	Private



Spezielle Methoden und Vererbung

- Einige Funktionen müssen verschiedene Dinge in der Basisklasse und in der abgeleiteten Klasse machen.
- Diese speziellen Funktionen können nicht vererbt werden.
- **Konstruktoren:** der Konstruktor der abgeleiteten Klasse muss andere Daten erstellen.
- **Assignment Operator:** In der abgeleiteten Klasse dieser Operator muss den abgeleiteten Daten Werte zuweisen.
- **Destruktoren**



Konstruktoren/Destruktoren bei Vererbung

- Konstruktoren und Destruktoren werden nicht geerbt.
- Konstruktoren in der abgeleiteten Klasse müssen einen Konstruktor der Basisklasse aufrufen.
- Wenn kein Konstruktor explizit aufgerufen wird, wird automatisch der Default-Konstruktor der Basisklasse aufgerufen.
- kein Default-Konstruktor → Compilerfehler.



Konstruktoren/Destruktoren bei Vererbung

- Wenn ein Objekt einer abgeleiteten Klasse erstellt wird, wird der Konstruktor der Basisklasse zuerst aufgerufen (und dann der Konstruktor der abgeleiteten Klasse).
- Der Destruktor der Basisklasse wird automatisch von der Destruktor der abgeleiteten Klasse aufgerufen.
- Wenn ein Objekt einer abgeleiteten Klasse zerstört wird, wird der Destruktor der abgeleiteten Klasse zuerst aufgerufen (und dann der Destruktor der Basisklasse).



Konstruktoren/Destruktoren bei Vererbung

Erzeugen/Anlegen:

1. Reservieren von benötigten Ressourcen für Variablen der Basisklasse;
2. Reservieren von benötigten Ressourcen für Variablen der abgeleiteten Klasse;
3. Ein Konstruktor wird ausgesucht und aufgerufen, um die Variablen der Basisklasse zu initialisieren;
4. Ein Konstruktor wird ausgesucht und aufgerufen, um die Variablen der abgeleiteten Klasse zu initialisieren.

Zerstörung:

1. Destruktoraufruf für die abgeleitete Klasse;
2. Destruktoraufruf für die Basisklasse.