

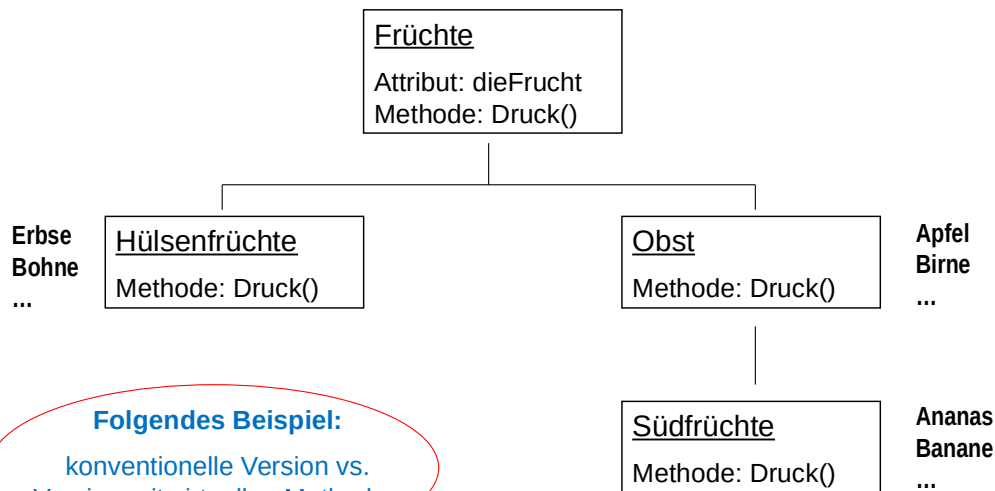
# Einführung in die Programmierung

Wintersemester 2020/21

## Kapitel 11: Virtuelle Methoden

M.Sc. Roman Kalkreuth  
 Lehrstuhl für Algorithm Engineering (LS11)  
 Fakultät für Informatik

### Klassenhierarchie



**Folgendes Beispiel:**  
 konventionelle Version vs.  
 Version mit virtuellen Methoden

### Vererbung bisher:

- Definition von Klassen basierend auf anderen Klassen
  - Übernahme (erben) von Attributen und Methoden
  - Methoden können überschrieben werden

↓

Bindung der Methoden an Objekte  
 geschieht **zur Übersetzungszeit!**

### jetzt:

Technik zur Bindung von Methoden an Objekte **zur Laufzeit**  
 → dynamische Bindung: **Polymorphismus**

### Konventionelle Version

```

class Frucht {
protected:
    string dieFrucht;
public:
    Frucht(char const *name);
    Frucht(string &name);
    void Druck();
};
    
```

Frucht.h

```

Frucht::Frucht(char const *name) :
dieFrucht(name) { }

Frucht::Frucht(string &name) :
dieFrucht(name) { }

void Frucht::Druck() {
    cout << "(F) "
        << dieFrucht << endl;
}
    
```

Frucht.cpp

Konventionelle Version

```
class HFrucht : public Frucht {
public:
    HFrucht(char const *name);
    void Druck();
};
```

Unterklasse von **Frucht**

```
class Obst : public Frucht {
public:
    Obst(char const *name);
    void Druck();
};
```

Unterklasse von **Frucht**

```
class SFrucht : public Obst {
public:
    SFrucht(char const *name);
    void Druck();
};
```

Unterklasse von **Obst**

Konventionelle Version: Testprogramm

```
int main() {
    Frucht *ruebe = new Frucht("Ruebe");
    ruebe->Druck();

    HFrucht *erbse = new HFrucht("Erbse");
    erbse->Druck();

    Obst *apfel = new Obst("Apfel");
    apfel->Druck();

    SFrucht *banane = new SFrucht("Banane");
    banane->Druck();
}
```

} 1. Teil

Ausgabe: **(F) Ruebe**  
**(H) Erbse**  
**(O) Apfel**  
**(S) Banane**

Konventionelle Version

```
HFrucht::HFrucht(char const *name) : Frucht(name) { }

void HFrucht::Druck() {
    cout << "(H) " << dieFrucht << endl;
}
```

```
Obst::Obst(char const *name) : Frucht(name) { }

void Obst::Druck() {
    cout << "(O) " << dieFrucht << endl;
}
```

```
SFrucht::SFrucht(char const *name) : Obst(name) { }

void SFrucht::Druck() {
    cout << "(S) " << dieFrucht << endl;
}
```

Konventionelle Version: Testprogramm

```
Frucht *f = new Frucht("Frucht");
f->Druck();

f = apfel; // jedes Obst ist auch Frucht
f->Druck();

Obst *o = new Obst("Obst");
o->Druck();

o = banane; // Suedfrucht ist auch Obst
o->Druck();
}
```

} 2. Teil

Ausgabe: **(F) Frucht**  
**(F) Apfel**  
**(O) Obst**  
**(O) Banane**

Merke:

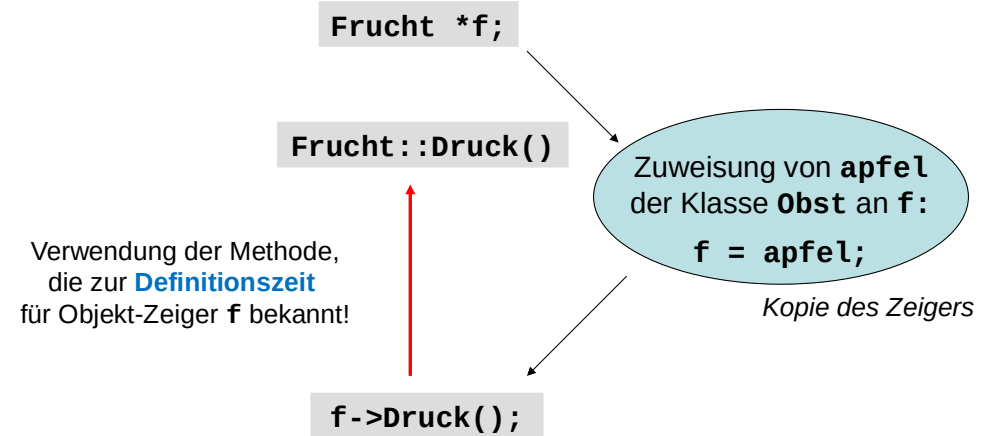
- Zuweisungen sind **entlang der Vererbungshierarchie** möglich
  - Objekt kann einem Objekt seiner Oberklasse zugewiesen werden
- Methoden sind (hier) **statisch** an Objekt gebunden
  - zur Übersetzungszeit bekannte Methode wird ausgeführt
  - Zuweisung eines Objekts einer abgeleiteten Klasse führt **nicht** zur Übernahme der überschriebenen Methoden der Unterklasse



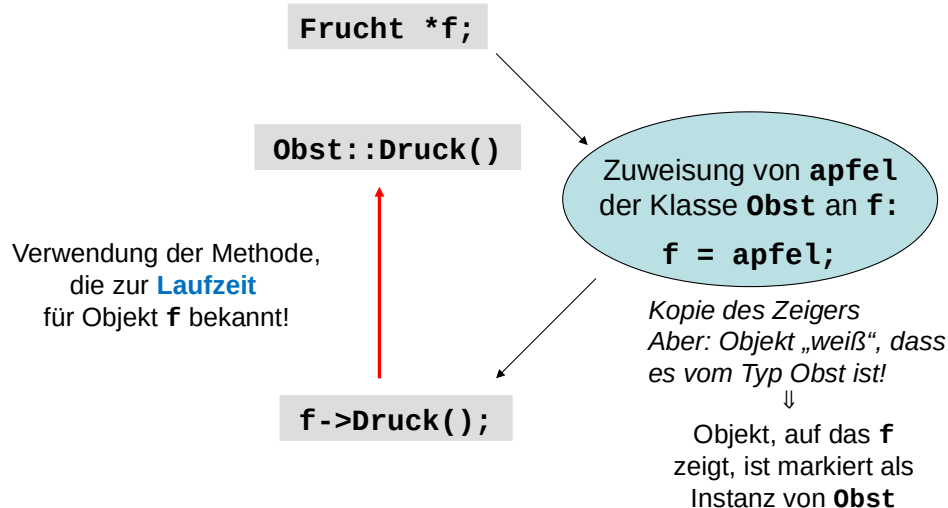
Wenn man das haben möchte, dann müssten die Methoden der Unterklasse **zur Laufzeit** (bei der Zuweisung) an das Objekt **gebunden** werden.

→ **dynamische Bindung**

Statische Methodenbindung



Dynamische Methodenbindung



Virtuelle Methoden

- sind Methoden, die zur Laufzeit (also dynamisch) gebunden werden sollen;
- werden in der Oberklasse durch Schlüsselwort **virtual** gekennzeichnet.

Wird eine **virtuelle Methode** in einer abgeleiteten Klasse **überschrieben**, so wird die Methode ausgewählt, die sich aus dem **Typ** des Objekts **zur Laufzeit** ergibt.

Version mit virtuellen Funktionen

```
class Frucht {
protected:
    string dieFrucht;
public:
    Frucht(char const *name);
    Frucht(string &name);
    virtual void Druck();
};
```

Ansonsten keine Änderungen im Code der konventionellen Version!

Kennzeichnung als **virtuelle Methode**:  
Instanzen von abgeleiteten Klassen suchen dynamisch die entsprechende Methode aus.

**Achtung: Zeiger notwendig!**

```
SFrucht *kiwi = new SFrucht("kiwi");
kiwi->Druck();

Obst obst("Obst statisch");
obst.Druck();

obst = *kiwi;
obst.Druck();
```

nur Daten-, keine Typkopie wie statische Bindung

Ausgabe: (S) kiwi  
(dynamisch, also mit virtual) (O) Obst statisch  
(O) kiwi

**dynamische Bindung funktioniert nur mit Zeigern oder Referenzen!**

Konsequenzen: Testprogramm mit virtuellen Methoden (nur 2. Teil)

```
Frucht *f = new Frucht("Frucht");
f->Druck();

f = apfel; // jedes Obst ist auch Frucht
f->Druck();

Obst *o = new Obst("Obst");
o->Druck();

o = banane; // Suedfrucht ist auch Obst
o->Druck();
}
```

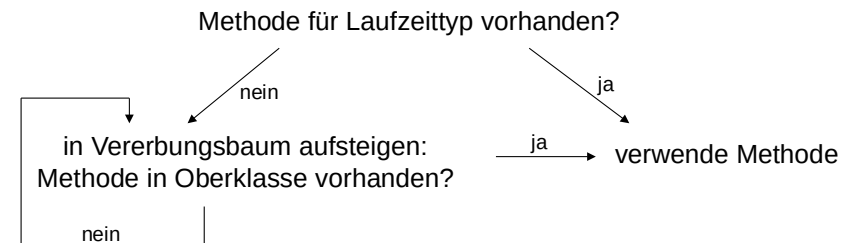
} 2. Teil

Ausgabe: (F) Frucht  
(dynamisch, also mit virtual) (O) Apfel  
(O) Obst  
(S) Banane

Ausgabe: (F) Frucht  
(statisch, also ohne virtual) (F) Apfel  
(O) Obst  
(O) Banane

Anmerkung:

Als virtuell gekennzeichnete Methode **muss nicht** in jeder abgeleiteten Klasse **redefiniert / überschrieben werden**.



Beispiel

```
class X {
public:
    virtual void Druck();
};
```

```
void X::Druck() {
    cout << "X";
}
```

```
class Y : public X {
public:
    void Druck();
};
```

```
void Y::Druck() {
    cout << "Y";
}
```

```
class Z : public Y { };
```

```
int main() {
    X *p[4] = { new X, new Y, new X, new Z };
    for (int i = 0; i < 4; i++) p[i]->Druck();
    cout << endl;
}
```

Ausgabe:

XYXY

dynamische Bindung!

Rein virtuelle Methoden

Annahme:

Wir wollen erzwingen, dass jeder Programmierer, der von unserer Basisklasse eine neue Klasse ableitet, eine bestimmte Methode implementiert bzw. bereitstellt.

Realisierung in C++:

1. Die Methode wird als virtuell (virtual) deklariert.
2. Bei der Deklaration wird hinter der Signatur =0 eingefügt.
3. Die Methode bleibt in dieser Klasse undefiniert.
  - ⇒ Die Erben müssen die Definition der Methode nachholen!

Beispiel

```
class X {
public:
virtual void Druck();
};
```

```
void X::Druck() {
    cout << "X";
}
```

```
class Y : public X {
public:
    void Druck();
};
```

```
void Y::Druck() {
    cout << "Y";
}
```

```
class Z : public Y { };
```

```
int main() {
    X *p[4] = { new X, new Y, new X, new Z };
    for (int i = 0; i < 4; i++) p[i]->Druck();
    cout << endl;
}
```

Ausgabe:

XXXX

statische Bindung!

Rein virtuelle Methoden / abstrakte Klassen

aus dem C++-Standard:

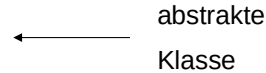
“An abstract class is a class that can be used only as a base class of some other class; no objects of an abstract class can be created except as subobjects of a class derived from it. A class is abstract if it has at least one pure virtual function.”



1. Klasse heißt abstrakt, wenn sie mindestens eine rein virtuelle Funktion hat.
2. Abstrakte Klassen können nicht instanziiert werden.
3. Abstrakte Klassen können als Basisklassen für andere Klassen benutzt werden.

Rein virtuelle Methoden

```
class AusgabeGeraet {
protected:
    bool KannFarben;
    Data data;
public:
    virtual void Farbdruck() = 0;
    void Drucke();
};
```



```
void AusgabeGeraet::Drucke() {
    if (KannFarben) Farbdruck();
    else cout << data;
}
```

Man kann rein virtuelle Methode verwenden, ohne dass Code vorhanden ist!

Wird ein Objekt einer abgeleiteten Klasse über einen Verweis / Zeiger auf die Basisklasse freigegeben, dann **muss** der **Destruktor in der Basisklasse virtuell** sein.

Warum?

Wenn nicht virtuell, dann findet die Bindung des Destruktors statisch zur Übersetzungszeit statt.

⇒ Immer Aufruf des Destruktors **der Basisklasse!**

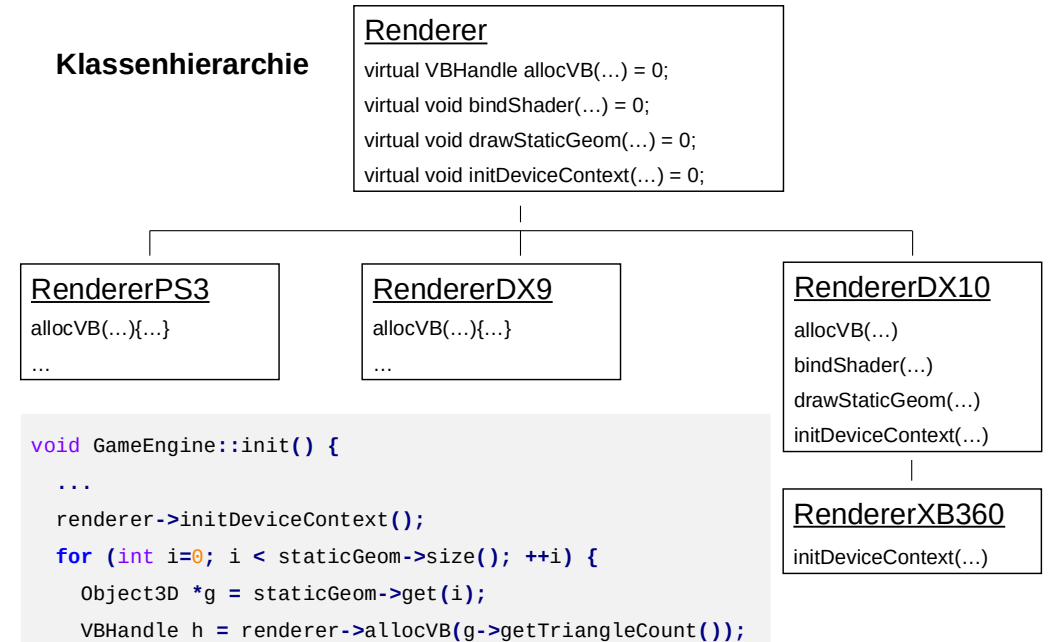
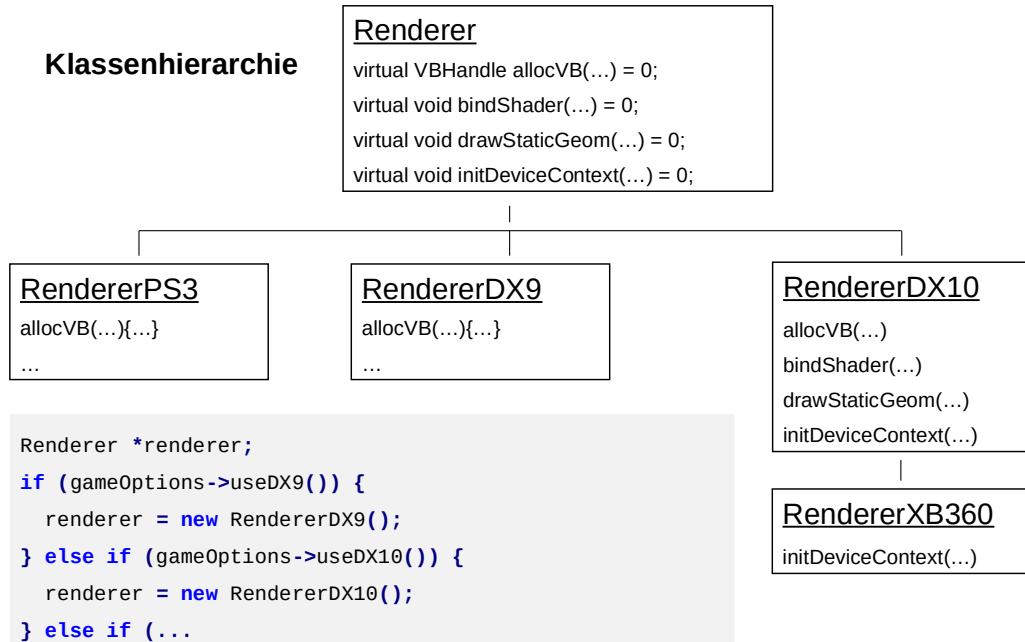
```
class Familie {
public:
    ~Familie() { cout << "D: Familie" << endl; }
};
class Sohn : public Familie {
    ~Sohn() { cout << "D: Sohn" << endl; }
};
class Tochter : public Familie {
    ~Tochter() { cout << "D: Tochter" << endl; }
};
int main() {
    Familie *fam[3] = { new Familie, new Sohn, new Tochter };
    delete fam[0]; delete fam[1]; delete fam[2];
    return 0;
}
```

Ausgabe: D: Familie  
D: Familie  
D: Familie

```
class Familie {
public:
    virtual ~Familie() { cout << "D: Familie" << endl; }
};
class Sohn : public Familie {
    ~Sohn() { cout << "D: Sohn" << endl; }
};
class Tochter : public Familie {
    ~Tochter() { cout << "D: Tochter" << endl; }
};
int main() {
    Familie *fam[3] = { new Familie, new Sohn, new Tochter };
    delete fam[0]; delete fam[1]; delete fam[2];
    return 0;
}
```

Ausgabe:

D: Familie  
D: Sohn  
D: Familie  
D: Tochter  
D: Familie



**Wofür machen wir das alles (Vererbung, virtuelle Funktionen, Zeiger, Rekursion)?**

Joel Spolsky – Back to basics:  
<http://www.joelonsoftware.com/articles/fog0000000319.html> verified: 18.12.2019

**“If you want to teach somebody something well, you have to start at the very lowest level.**

**It's like Karate Kid.**

**Wax On, Wax Off. Wax On, Wax Off.**

**Do that for three weeks.**

**Then Knocking The Other Kid's Head off is easy.”**