



Programmieren 3

C++

Vorlesung 07: Polymorphie, Typumwandlung

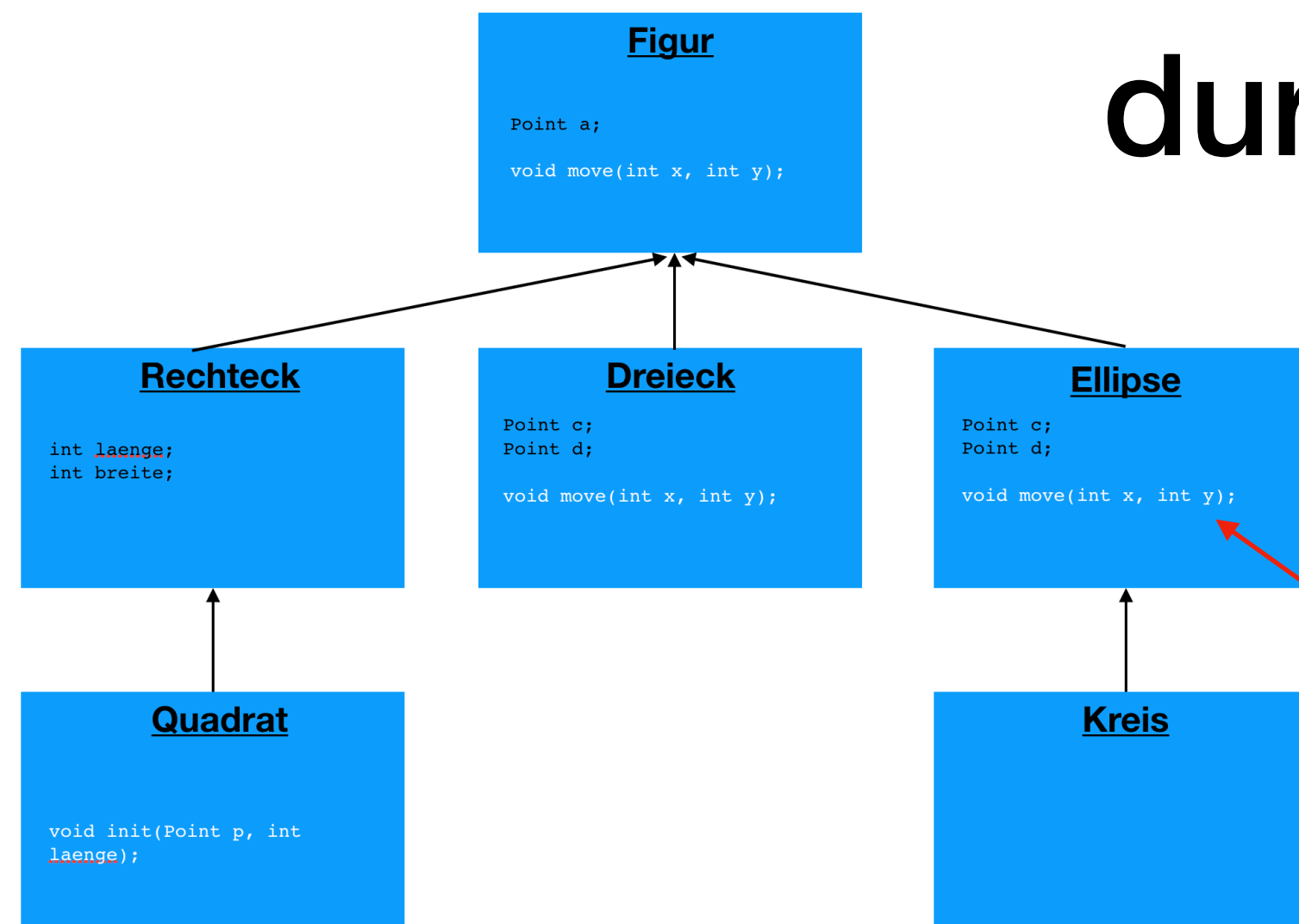
Prof. Dr. Dirk Kutscher

Dr. Olaf Bergmann

Wiederholung

Objektorientierte Programmierung

Wiederverwendung durch Vererbung



1. Funktionen von Basis-Klassen wiederverwenden

- z.B. `Ellipse::move` in `Kreis`

2. Funktionen, die auf Basisklassen arbeiten, können auch auf abgeleitete Klassen angewendet werden → *Referenzparameter*

```
class Figur {
    Point a;
public:
    void move(int x, int y);
};

class Rechteck: public Figur {
    int laenge;
    int breite;
};

void moveAndDraw(Figur &f) {
    f.move(10,10);
    // draw...
}

int main() {
    Figur fi;
    Rechteck re;

    moveAndDraw(fi);
    moveAndDraw(re);
}
```

Zugriffsrechte

```
class Oberklasse {  
private:  
    int oberklassePriv;  
    void privateFunktionOberklasse();  
protected:  
    int oberklasseProt;  
public:  
    int oberklassePubl;  
    void publicFunktionOberklasse();  
};
```

// Voreinstellung

Zugriffsrecht in der Basisklasse	Zugriffsrecht in einer abgeleiteten Klasse
private protected public	kein Zugriff protected public

// Oberklasse wird mit der Zugriffsbezeichnung public vererbt

```
class AbgeleiteteKlasse : public Oberklasse {  
    int abgeleiteteKlassePriv;  
public:  
    int abgeleiteteKlassePubl;  
    void publicFunktionAbgeleiteteKlasse() {  
        oberklassePriv = 1;  
        // in einer abgeleiteten Klasse zugreifbar  
        oberklasseProt = 2;  
        // generell zugreifbar  
        oberklassePubl = 3;  
    }  
};
```

Man kann auch `private` oder `protected` erben
— meistens nicht sinnvoll

// Fehler: nicht zugreifbar

```
int main() {  
    AbgeleiteteKlasse objekt;  
    int m = objekt.oberklassePubl;  
    m = objekt.oberklasseProt;
```

// Fehler: nicht zugreifbar


```

class Ort {
    // ...
};

class GraphObj { // Version 1
public:
    GraphObj(Ort einOrt) // allgemeiner Konstruktor
        : referenzkoordinaten{einOrt} {}

    // Bezugspunkt ermitteln
    Ort bezugspunkt(void) const {
        return referenzkoordinaten;
    }

    // alten Bezugspunkt ermitteln und gleichsetzen
    Ort bezugspunkt(Ort n0) {
        Ort temp {referenzkoordinaten};
        referenzkoordinaten = n0;
        return temp;
    }

    // Koordinatenabfrage
    int getX(void) const {
        return referenzkoordinaten.getX();
    }
    int getY(void) const {
        return referenzkoordinaten.getY();
    }

    // Standardimplementation
    double flaeche(void) const {
        return 0.0;
    }

private:
    Ort referenzkoordinaten;
};

// Die Entfernung zwischen 2 GraphObj-Objekten ist hier als Entfernung ihrer
// Bezugspunkte (überladene Funktion) definiert.
inline double entfernung(GraphObj g1, GraphObj g2) {
    return entfernung(g1.bezugspunkt(), g2.bezugspunkt());
}

```

```

class Rechteck : public GraphObj {
public:
    Rechteck(Ort ort, int h, int b)
        : GraphObj{ort}, dieHoehe{h}, dieBreite{b} {}

    double flaeche(void) const {
        return dieHoehe * dieBreite;
    }

private:
    int dieHoehe;
    int dieBreite;
};

```

**Funktionen
überschreiben**

**Kann zur Übersetzungszeit
entschieden werden**

Polymorphie:

Zur Laufzeit „das Richtige“ tun

```
class A {
public:
    void ausgabe(void) {cout << "A" << endl;}
};

class B: public A {
public:
    void ausgabe(void) {cout << "B" << endl;}
};

void ausgeben(A &obj) {
    obj.ausgabe();
}

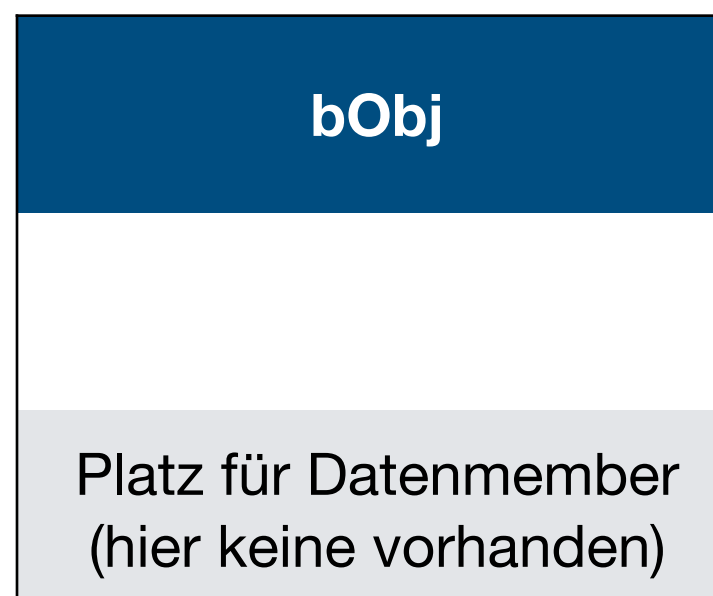
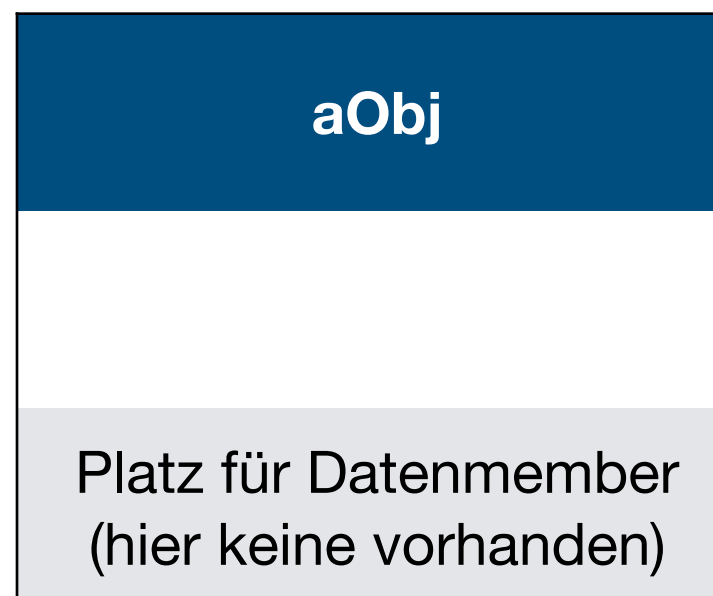
int main() {
    A aObj;
    B bObj;

    ausgeben(aObj);
    ausgeben(bObj);
}
```

- Funktion ausgeben:
 - „A“ für Objekte vom Typ A
 - „B“ für Objekte vom Typ B
- Dafür muss **zur Laufzeit** entschieden werden, welche ausgabe-Funktion auf einem Objekt wirklich ausgeführt wird

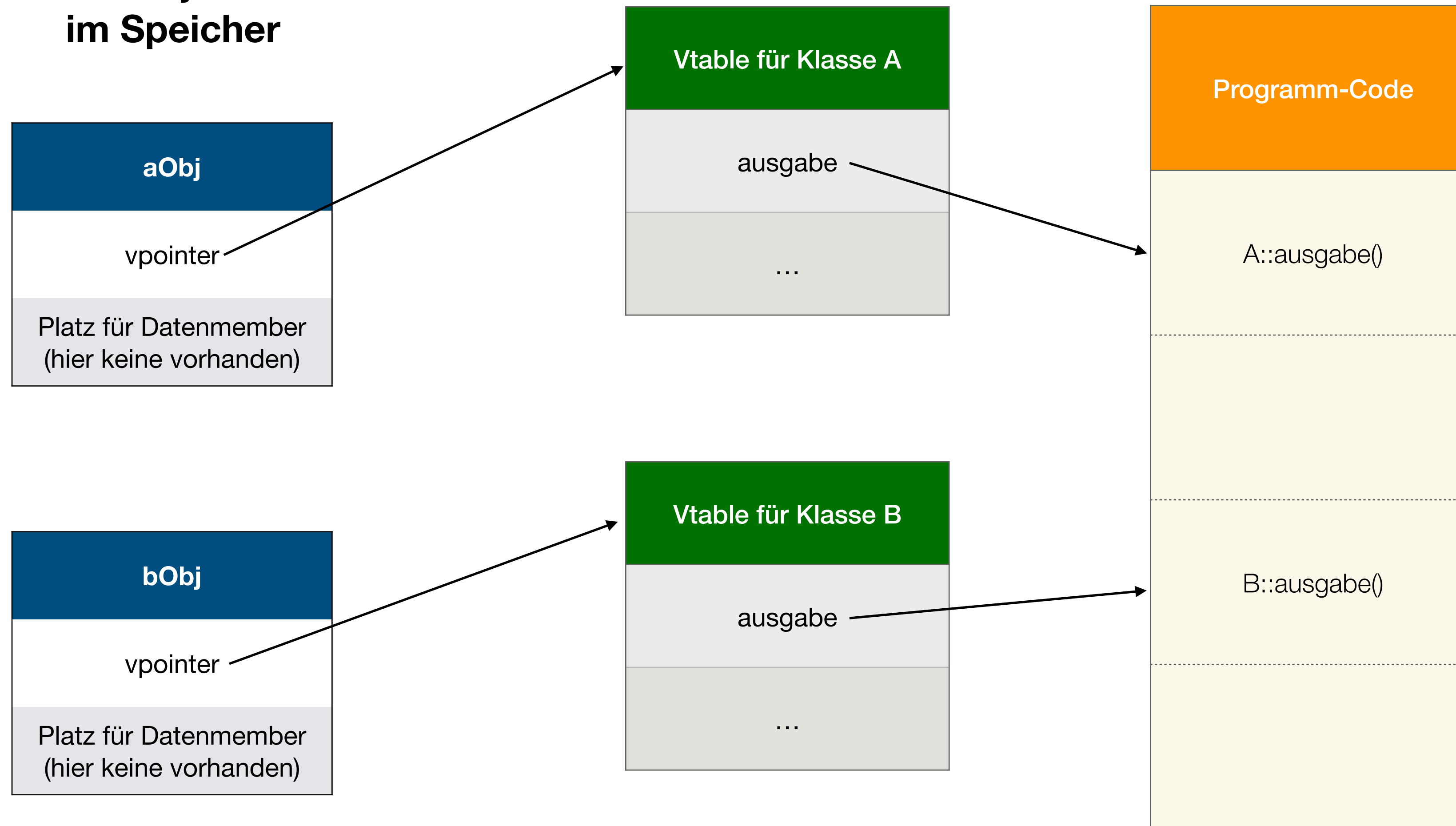
Polymorphie mit virtuellen Funktionen

**Repräsentation
von Objekten
im Speicher**



Polymorphie mit virtuellen Funktionen

Repräsentation
von Objekten
im Speicher



Beim Aufruf von `ausgabe()` auf einem Objekt vom Typ A oder B wird *zur Laufzeit* ermittelt, welche Funktion wirklich ausgeführt werden muss.

Polymorphie mit Virtuellen Methoden

```
class A {  
public:  
    virtual void ausgabe(void) {cout << "A" << endl;}  
};
```

bewirkt Eintrag in Tabelle
virtueller Methoden (Vtable)

```
class B: public A {  
public:  
    void ausgabe(void) override {cout << "B" << endl;}  
};
```

Optional: Compiler prüft bei
Übersetzung, ob virtuelle Methode
mit dieser Signatur definiert ist.

```
void ausgeben(A &obj) {  
    obj.ausgabe();  
}
```

```
int main() {  
    A aObj;  
    B bObj;  
  
    ausgeben(aObj);  
    ausgeben(bObj);  
}
```

Virtuelle Funktionen

```
class A {
public:
    virtual void ausgabe(void)
        {cout << "A" << endl;}
};

class B: public A {
public:
    void ausgabe(void) override
        {cout << "B" << endl;}
};

void ausgeben(A &obj) {
    obj.ausgabe();
}

int main() {
    A aObj;
    B bObj;

    ausgeben(aObj);
    ausgeben(bObj);
}
```

- **Überschreiben von Funktionen gleicher Signatur bei Vererbung**
- **Beim Aufruf wird der dynamische Typ des Objekts ermittelt (z.B. A oder B)**
 - Richtige Funktion wird über vpointer und Vtable gefunden
- **Motivation: Basisklasse definiert Aufrufchnittstelle (z.B. ausgabe)**
 - Abgeleitete Klassen können neues Verhalten definieren
 - Generische Funktionen (wie `ausgeben`) können dann auf allen Objekten der Basisklasse oder abgeleiteten Klassen operieren
 - Trotzdem kann für jedes Objekt die spezialisierte Funktion aufgerufen werden

Abstrakte Klassen

- **Rein virtuelle Funktionen (Pure Virtual Functions)**
- Wird mit `virtual ... = 0` deklariert, aber nicht definiert
- Abgeleitete Klassen *müssen* dann diese Funktion definieren
- Objekte vom Typ der Basisklasse (hier `GraphObj`) können nicht instanziiert werden, daher **Abstrakte Klassen**

```
class Ort {
    // ...
};

class GraphObj { // Version 1
public:
    GraphObj(Ort einOrt) // allgemeiner Konstruktor
        : referenzkoordinaten{einOrt} {}

    // Bezugspunkt ermitteln
    Ort bezugspunkt(void) const {
        return referenzkoordinaten;
    }

    // alten Bezugspunkt ermitteln
    // und gleichzeitig neuen wählen
    Ort bezugspunkt(Ort n0) {
        Ort temp {referenzkoordinaten};
        referenzkoordinaten = n0;
        return temp;
    }

    // Koordinatenabfrage
    int getX(void) const {
        return referenzkoordinaten.getX();
    }
    int getY(void) const {
        return referenzkoordinaten.getY();
    }

    // Standardimplementation
    virtual double flaeche(void) = 0;

private:
    Ort referenzkoordinaten;
};
```

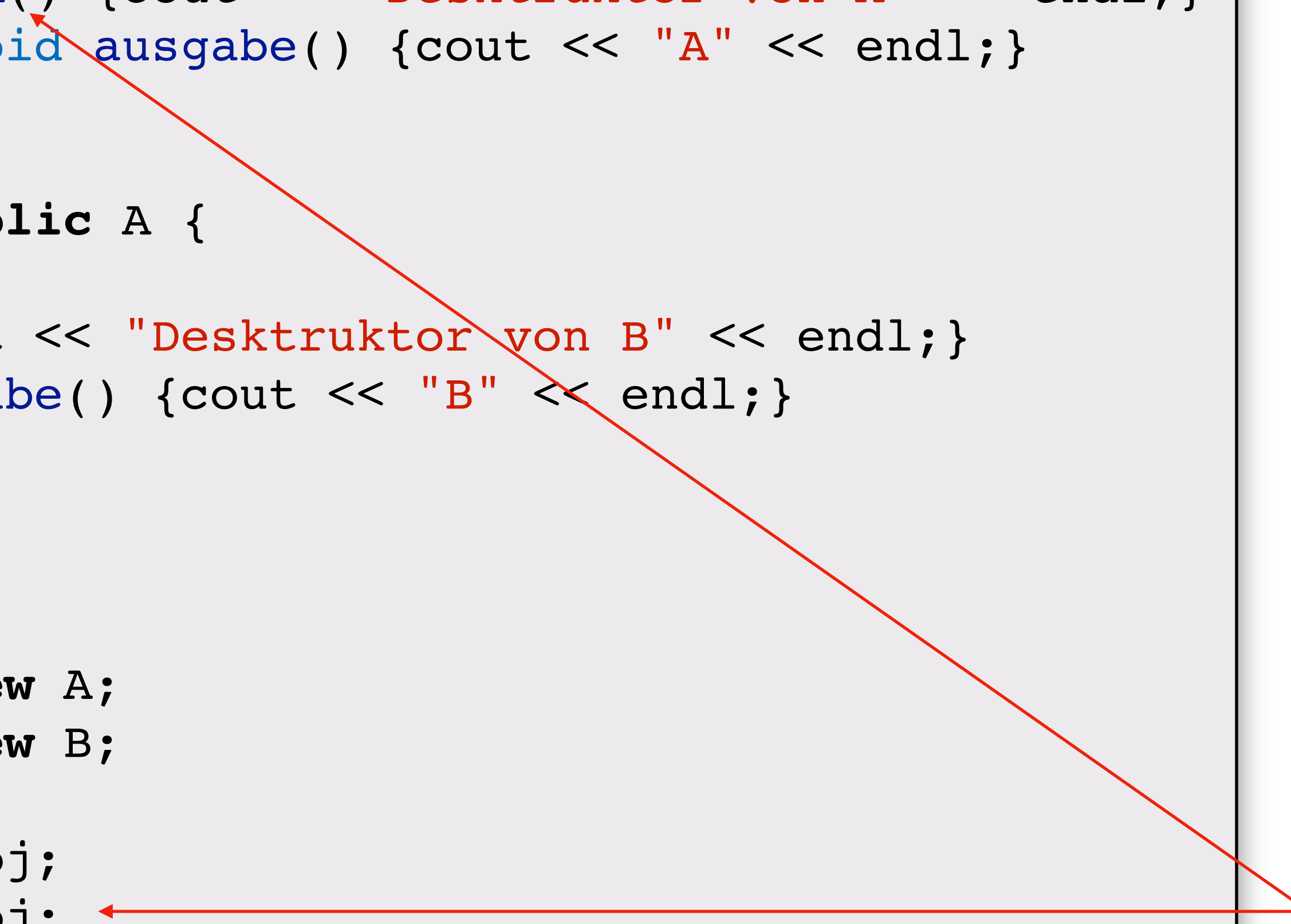
```
class Strecke : public GraphObj {
public:
    Strecke(Ort ort1, Ort ort2)
        : GraphObj{ort1},
          endpunkt{ort2} {}

    auto laenge(void) const {
        return entfernung(bezugspunkt(), endpunkt);
    }

    virtual double flaeche(void) const override {
        return 0.0;
    }
};
```

Virtueller Destruktor

```
class A {  
public:  
    virtual ~A() {cout << "Destruktor von A" << endl;}  
    virtual void ausgabe() {cout << "A" << endl;}  
};  
  
class B: public A {  
public:  
    ~B() {cout << "Destruktor von B" << endl;}  
    void ausgabe() {cout << "B" << endl;}  
};  
  
int main() {  
    A* aObj=new A;  
    A* bObj=new B;  
  
    delete aObj;  
    delete bObj;  
}
```



Container in C++ (Stdlib)

Sequence containers

Sequence containers implement data structures which can be accessed sequentially.

array (C++11)	static contiguous array (class template)
vector	dynamic contiguous array (class template)
deque	double-ended queue (class template)
forward_list (C++11)	singly-linked list (class template)
list	doubly-linked list (class template)

Associative containers

Associative containers implement sorted data structures that can be quickly searched ($O(\log n)$ complexity).

set	collection of unique keys, sorted by keys (class template)
map	collection of key-value pairs, sorted by keys, keys are unique (class template)
multiset	collection of keys, sorted by keys (class template)
multimap	collection of key-value pairs, sorted by keys (class template)

std::map

```
#include <map>
#include <string>
#include <iostream>
```

```
using namespace std;
```

```
int main() {
    map<string, int> preis;
```

```
    preis["Toastbrot"] = 2;
    preis["Schokolade"] = 3;
    preis["Kaffee"] = 10;
```

```
    cout << preis["Kaffee"] << endl;
```

```
    pair<string, int> ersterPreis = *preis.begin();
```

```
    cout << ersterPreis.first << ": " << ersterPreis.second << endl;
```

```
    for(auto p = preis.begin(); p != preis.end(); p++) { // gibt alle Paare aus
        cout << p->first << ": " << p->second << endl;
    }
}
```

Mal angucken...

<https://en.cppreference.com/w/cpp/container/map>
<https://en.cppreference.com/w/cpp/utility/pair>

besser:

```
for (auto const &p: preis) {
    cout << p.first << ": "
        << p.second << endl;
}
```

Übung zur Vertiefung

- Re-Implementieren Sie SimpleString als abgeleitete Klasse von MyString
 - Rein virtuelle Funktionen implementieren
 - Außerdem: Operatoren definieren:
 - `operator+=` („Addieren/Anhängen und Zuweisen“)
 - `operator==` (und andere Vergleichsoperatoren)
- Testen

```
1 #include <string>
2 #include <vector>
3 #include <iostream>
4
5 using namespace std;
6
7 class MyString {
8
9 public:
10     virtual ~MyString(void) {};
11     virtual std::string to_string(void) const = 0; // return representation as std::string
12
13     virtual int len(void) const = 0; // return current string length -- ACHTUNG: sollte const sein
14
15     virtual int find(const MyString &s) const = 0; // find, return index if found, 0 if not found
16
17     virtual void clear(void) = 0; // make this string empty
18     virtual void print(void) const = 0; // print string to std::cout
19 };
```

Zusammenfassung

- **Virtuelle Funktionen**

- Funktionsnamen von Basisklassen bei abgeleiteten Klassen wiederverwenden
- Das heißt, die Funktion in der abgeleiteten Klasse neu definieren
- Zur Laufzeit (über Typinformationen im Objekt) die passende Funktion bestimmen und ausführen

- **Rein virtuelle Funktionen (*pure virtual functions*)**

- **Basisklasse** gibt nur Interface vor, definiert aber gar keine Funktion
- Von solchen Klassen kann man keine Objekte erzeugen (**Abstrakte Basisklassen**)
- Abgeleitete Klassen *müssen* dann die entsprechenden Funktionen definieren

- **Zuweisungs- und Vergleichsoperationen**

- Müssen in der Regel auf Eigenschaften der abgeleiteten Klasse zugreifen
- Schwierig, dies ohne Verrenkungen zu vermeiden
- Daher besser *nicht* als virtuelle Funktionen definieren

Was, wenn ich trotzdem operator= in abgeleiteten Klassen verwenden möchte?

- Abgeleitete Klasse MyNewString
- verfügt über eigene Daten-Member
- Zuweisungsoperator sollte diese kopieren, aber dann auch die Daten-Member der Basisklasse

```
111 class MyNewString: public SimpleString {  
112     int wordCount;  
114  
115 public:  
116     MyNewString(void):wordCount(0) {};  
117     const MyNewString &operator=(const MyNewString &rhs);  
118  
119 };
```

Was, wenn ich trotzdem `operator=` in abgeleiteten Klassen verwenden möchte?

```
27 class SimpleString: public MyString {
28
29 protected:
30     const SimpleString &assign(const SimpleString &s)
31     {buffer = s.buffer; theLength = s.theLength; return *this;}
32
33 public:
34     SimpleString(void) : theLength(0){};
35     SimpleString(const char *initString);
36
37     int len(void) const {return theLength;};
38     std::string to_string(void) const;
39     int find(const MyString &s) const;
40     void clear(void) {buffer.clear(); theLength=0;}
41     void print(void) {cout << to_string();};
42
43     const SimpleString &operator+=(const SimpleString &addedString);
44     bool operator==(const SimpleString &s) const; // non-virtual
45
46 private:
47     vector<char> buffer;
48     int theLength;
49 };
```

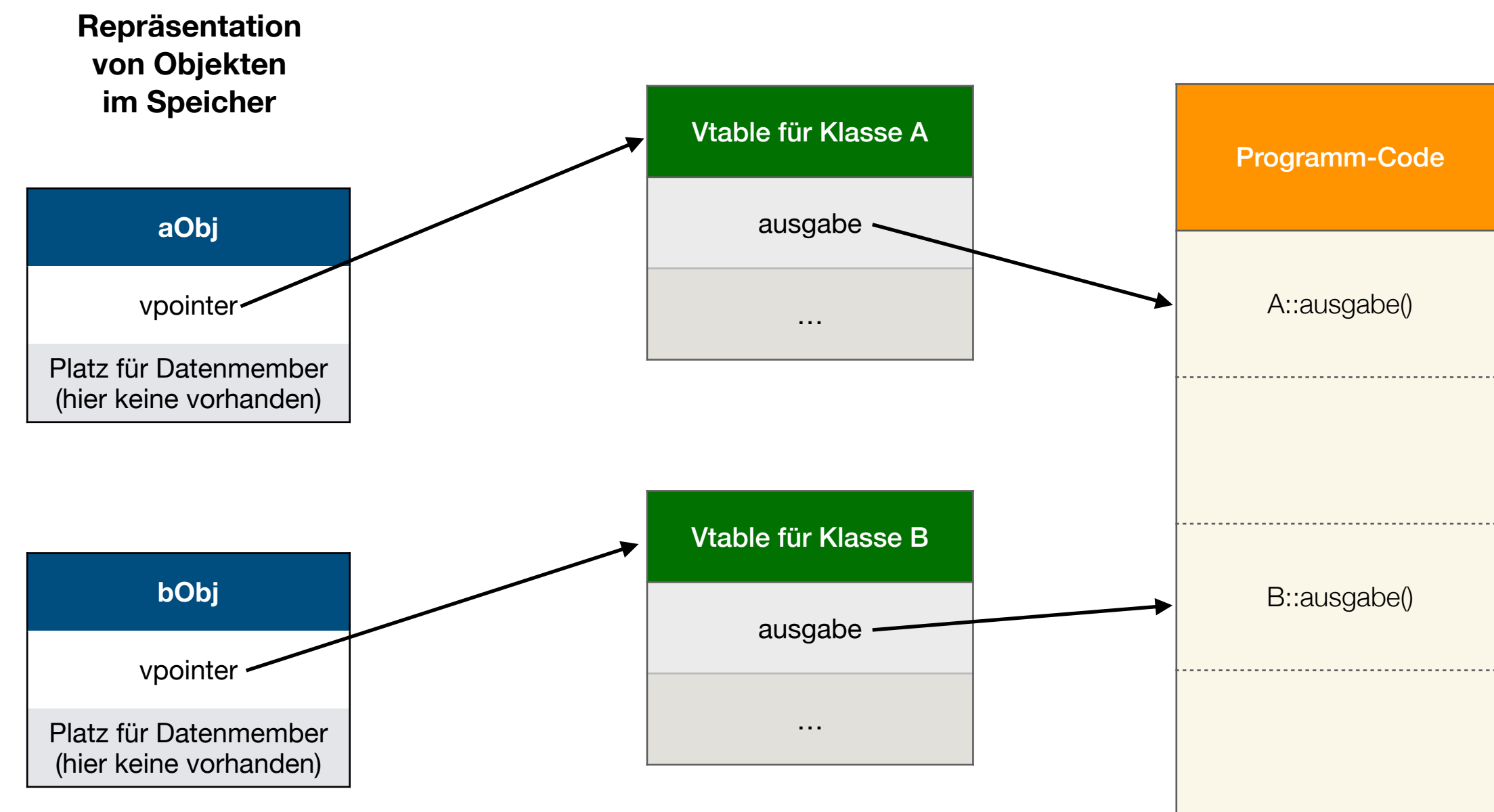
```
111 class MyNewString: public SimpleString {
112
113     int wordCount;
114
115 public:
116     MyNewString(void) : wordCount(0){};
117     const MyNewString &operator=(const MyNewString &rhs);
118
119 };
```

- Kann man über Konventionen lösen
- Beispiel: protected-Funktion `assign` in der Basisklasse
- Kann in der abgeleiteten Klassen explizit aufgerufen werden
- Entsprechende Konventionen müsste man für eine Klassenhierarchie festlegen
 - z. B. dass alle Klassen eine Funktion `assign` definieren sollten
- **Achtung: hier ist `operator=` eigentlich überflüssig**
 - Wenn man diesen weglässt, würde C++ automatisch alle Member-Variablen der aktuellen Klassen und der Basisklassen kopieren
 - Man muss `operator=` nur definieren, wenn man ein anderes Verhalten bewirken möchte (z. B. dynamisch Speicher reservieren)

```
122 const MyNewString&
123 MyNewString::operator=(const MyNewString &rhs) {
124     wordCount = rhs.wordCount; // Member von MyNewString kopieren
125     SimpleString::assign(rhs); // SimpleString-Teil kopieren
126     return *this;
127 }
```


Typ-Informationen zur Laufzeit verwenden

- Virtuelle Funktionen machen das „automatisch“
- Man kann in C++ auch als Programmierer auf Typ-Informationen zugreifen
- Könnte man theoretisch in Methoden abgeleiteter Klassen verwenden, um herauszubekommen, von welchem Typ ein übergebenes Objekt wirklich ist
- Achtung: sparsam verwenden
- Meistens benötigt man das nicht



Typ-Informationen zur Laufzeit verwenden

```
• #include <iostream>
  #include <typeinfo>

  class Base {
  public:
    virtual const char *type(void) const {
      return typeid(*this).name();
    }
  };

  class Derived : public Base {};

  int main() {
    Base b;
    Derived d;
    std::cout << b.type() << " " << d.type() << std::endl;
  }
```

(plattformabhängige) Repräsentation
als Zeichenkette (nicht standardisiert)

Liefert Typ-Information zur Laufzeit.

- Achtung: sparsam verwenden
- Meistens benötigt man das nicht

Typ-Infos zur Laufzeit

```
6 class Fahrrad {
7     string marke;
8 public:
9     virtual void printBike() {cout << "Marke...";}
10 };
11
12 class EBike: public Fahrrad {
13     int kapazitaet;
14 public:
15     virtual void printBike() {cout << "EBike-Marke...";}
16 };
17
18 class MTB: public Fahrrad {
19     int reifengroesse;
20 public:
21     virtual void printBike() {cout << "MTB-Marke...";}
22 };
```

```
bash-3.2$ ./printtype
EBike
```

```
25 void printType(const Fahrrad* fp) {
26     const type_info& fT(typeid(Fahrrad));
27     const type_info& eT(typeid(EBike));
28     const type_info& mT(typeid(MTB));
29     string typeResult("no type");
30
31     if(typeid(*fp)==fT)
32         typeResult="Fahrrad";
33     else if(typeid(*fp)==eT)
34         typeResult="EBike";
35     else if(typeid(*fp)==mT)
36         typeResult="MTB";
37
38     cout << typeResult << endl;
39 }
```

```
41 int main() {
42     Fahrrad* ebike=new EBike;
43
44     printType(ebike);
45
46     return 0;
47
48 }
```

<https://en.cppreference.com/w/cpp/language/typeid>

Typumwandlung

- Wir kennen Typumwandlung aus C (Casting), z. B.:

```
int main() {  
    int i=42;  
    int *iptr;  
  
    void *ptr;  
  
    ptr = &i;  
  
    iptr = (int*)ptr;  
}
```

```
void f(int i) {  
    unsigned int ui = (unsigned int)i;  
}
```

Typumwandlung in C: Probleme

- Keine Kontrolle über Casting-Operationen
- Illegales (gefährliches) Casting immer möglich
- Schwierig, harmloses von gefährlichem zu unterscheiden
- Syntax ohnehin nicht optimal (Klammerschreibweise wird auch für viele andere Dinge benutzt...)

```
void badCast(void *ptr) {  
    int *zahl = (int*)ptr;  
}  
  
void otherFunc() {  
    char c='A';  
  
    badCast(&c);  
}
```

Zugriff *zahl kann
Programmabsturz
verursachen (→ Bus Error)

Casting in C++

- Programmierer soll weiterhin Freiheit haben
- Aber: harmloses von gefährlichem Casting unterscheidbar machen
- Einige Cast-Operationen vom Compiler überprüfbar machen
- Daher unterschiedliche Cast-Operationen definiert:
 - `static_cast`: Typumwandlung zur Übersetzungszeit
 - `dynamic_cast`: Typumwandlung zur Laufzeit
 - `const_cast`: `const`-Eigenschaften entfernen
 - `reinterpret_cast`: Datentyp beliebig neu interpretieren

static_cast

Scoped Enum

```
enum class Wochentag {sonntag, montag, dienstag, mittwoch,  
                     donnerstag, freitag, samstag  
} heute = Wochentag::dienstag;
```

```
int i = static_cast<int>(Wochentag::dienstag);  
heute = i; // Fehler, Datentyp inkompatibel  
heute = static_cast<Wochentag>(i); // erlaubt!
```

Ab C++23 auch:

```
i = to_underlying(heute);
```

- Typumwandlungen durchführen oder rückgängig machen
- Zur Übersetzungszeit wird geprüft, ob Typen kompatibel sind
- Auch bei Vererbung anwendbar:

```
GraphObj g(Ort(3, 17));  
Strecke s(Ort(3, 17), (Ort(0, 0))); // Strecke ist von GraphObj abgeleitet  
GraphObj *pg;  
Strecke *ps {&s};  
pg = ps; // bekannte implizite Konversion  
ps = pg; // verboten!  
ps = (Strecke*) pg; // gefährlicher C-Stil!  
ps = static_cast<Strecke*> (pg); // nur richtig, falls pg auf ein Strecke-Objekt zeigt
```

Besser mit dynamic_cast...

dynamic_cast

```
class A {  
public:  
    virtual void foo();  
    void f();  
};  
  
class B: public A {  
public:  
    void g();  
};  
  
void test(A* obj) {  
    B* bobj = dynamic_cast<B*>(obj);  
    if(bobj != 0) {  
        bobj->g();  
    } else {  
        // Fehler  
    }  
}
```

Auch mit Referenzen:

```
B &bobj = dynamic_cast<B&>(ref);
```

→ kann Exception `bad_cast` auslösen

dynamic_cast

- `dynamic_cast<T> (Ausdruck)`
- Typüberprüfung findet zur Laufzeit statt
 - Auf der Basis der dynamischen Typ-Information
 - Basisklasse muss eine virtuelle Funktion haben
- Typ *T* muss ein Zeiger oder eine Referenz auf eine Klasse sein
- Falls das Argument *Ausdruck* ein Zeiger ist, der nicht auf ein Objekt vom Typ *T* (oder abgeleitet von *T*) zeigt, wird als Ergebnis 0 (Null-Pointer) zurückgegeben (Vorsicht!)
- Falls das Argument *Ausdruck* eine Referenz ist, die nicht auf ein Objekt vom Typ *T* (oder abgeleitet von *T*) verweist, wird eine Ausnahme (Exception) vom Typ `bad_cast` erzeugt.

const_cast

- `const`-Eigenschaft: signalisiert dem Compiler: Darf man nicht ändern
 - Hilfreich, wenn man unbeabsichtigtes Ändern automatisch verhindern lassen möchte
 - Bei Parameter-Übergabe übergibt man oft `const`-Referenzen (z. B. `const string &`): Effizient und trotzdem sicher
- Manchmal möchte man aber doch ein Objekt verändern können, das eigentlich `const` ist

```
const int i {100};  
const int *ip {&i};  
*ip = 0;                                     // geht nicht  
int *iq {const_cast<int*>(&i)};              // explizite Typumwandlung  
*iq = 0;                                     // Wert von i wird geändert!
```

Nur in begründeten Ausnahmefällen!

reinterpret_cast

```
#include <iostream>

int main() {
    double d;
    std::cout.write(reinterpret_cast<char*>(&d), sizeof(d));
}
```

- Typ-Umwandlung erzwingen
- Zum Beispiel, wenn man mit Byte-Arrays umgeht
- Maximal „brutaler“ Cast
- Sollte nur verwendet werden, wenn die anderen Cast-Operationen nicht anwendbar sind

Hinweise zu `typeid()` und Cast-Operationen

- Sparsam verwenden!
- Andauernde Abfragen von Typen deuten auf Probleme in der Programmstruktur hin
- Viele Laufzeit-Entscheidungen kann man dem Programm selbst überlassen: virtuelle Funktionen

```

class Fahrrad {
    string marke;
public:
    virtual void printBike() {cout << "Marke...";}
};

class EBike: public Fahrrad {
    int kapazitaet;
public:
    virtual void printBike() {cout << "EBike-Marke...";}
};

class MTB: public Fahrrad {
    int reifengroesse;
public:
    virtual void printBike() {cout << "MTB-Marke...";}
};

Fahrrad* copyFahrrad(const Fahrrad* fp) {
    const type_info& fT(typeid(Fahrrad));
    const type_info& eT(typeid(EBike));
    const type_info& mT(typeid(MTB));
    Fahrrad* res;

    if(typeid(*fp)==fT)
        res=new Fahrrad(*fp);
    else if(typeid(*fp)==eT)
        res=new EBike(*dynamic_cast<const EBike*>(fp));
    else if(typeid(*fp)==mT)
        res=new MTB(*dynamic_cast<const MTB*>(fp));

    return res;
}

```

Hässlich!


```

#include <iostream>
#include <string>

using namespace std;

class Fahrrad {
    string marke;
public:
    virtual void printBike(void) const {cout << "Marke...";}
    virtual Fahrrad *clone(void) const {return new Fahrrad(*this);}
};

class EBike: public Fahrrad {
    int kapazitaet;
public:
    void printBike(void) const override {cout << "EBike-Marke...";}
    Fahrrad *clone(void) const override {return new EBike(*this);}
};

class MTB: public Fahrrad {
    int reifengroesse;
public:
    void printBike(void) const override {cout << "MTB-Marke...";}
    Fahrrad *clone(void) const override {return new MTB(*this);}
};

Fahrrad *copyFahrrad(const Fahrrad *fp) {
    return fp->clone();
}

```