Informatik II

Oliver Jack

Fachhochschule Jena
Fachbereich Elektrotechnik und Informationstechnik

Sommersemester 2010

Inhalt

Vorlesung 8. Vererbung Teil 2

Vererbung

- Virtuelle Funktionen
- Abstrakte Klassen
- Mehrfachvererbung
- Virtuelle Vererbung
- Funktionsmaskierung
- Konstruktoren und Destruktoren in abgeleiteten Klassen
- Initialisierung
- Konstruktionsschema für abstrakte Klassen

Zusammenfassung

8. Vererbung Teil 2

Vorige Vorlesung

- Konzept der Vererbung
- UML-Klassendiagramme
- Wiederverwendung am Beispiel binärer Suchbaum

Heutige Vorlesung

- Virtuelle Funktionen
- Abstrakte Klassen
- Mehrfachvererbung
- Virtuelle Vererbung
- Initialisierung bei abgeleiteten Klassen
- Konstruktionsschema für abstrakte Klassen

Lernziele dieser Vorlesung

- Vertieftes Verständnis von Klassenhierarchien
- Verständnis von abstrakten Klassen
- Kenntnis der Konstruktion von abstrakten Klassen

Zeigerregel

- Abgeleitete Klassen sind bezüglich Zeiger kompatibel zur Basisklasse
- Ein Zeiger auf eine Basisklasse kann auf davon abgeleitete Klassen zeigen
- Ein Zeiger auf eine abgeleitete Klasse kann nicht auf die Basisklasse oder eine andere davon abgeleitete Klasse zeigen

Polymorphie und virtuelle Funktionen

- Virtuelle Funktionen erlauben Polymorphie von Funktionen
- Eine virtuelle Funktion wird in der Basisklasse deklariert und in abgeleiteten Klassen neu definiert
- Der Prototyp der neu definierten Funktion muss die selbe Struktur wie die virtuelle Funktion in der Basisklasse haben (gleiche Parameter, gleicher Datentyp des Rückgabewerts)
- Beim Aufruf einer virtuellen Funktion über einen Zeiger auf die Basisklasse wird die aktuelle Funktion durch den Typ, auf den der Zeiger zeigt, bestimmt und nicht durch die Basisklasse
- Dadurch wird zur Laufzeit die auszuführende Funktion bestimmt, und nicht zur Übersetzungszeit (Laufzeitbindung)

Virtuelle Funktionen

virtual *datentyp funktionsbezeichner*(*parameterdeklarationen*)
{
    anweisungen
}

- Die Eigenschaft, virtuell zu sein, wird durch die Vererbungshierarchie weitergereicht
- Ist eine Funktion in der Basisklasse als virtuell deklariert, so ist sie in allen (auch mehrstufig) abgeleiteten Klassen virtuell, ohne dort als virtuell deklariert zu sein

Beispiel Virtuelle Funktion...

```
#include <iostream>
using namespace std;
class basis {
public:
    int i;
    virtual void print_i ()
    {
      cout << i << "␣in␣basis" << endl;
    }
};
class abgeleitet: public basis {
public:
    void print_i ()
    {
      cout << i << "␣in␣abgeleitet" << endl;
    }
};
```

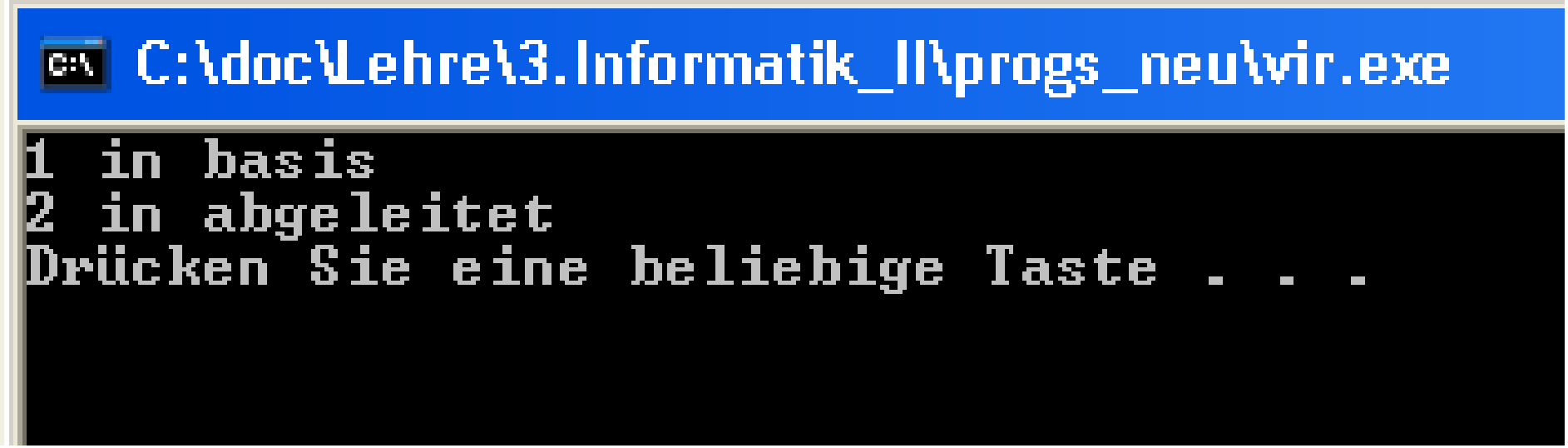
...Beispiel Virtuelle Funktion...

```
int main ()
{
    basis b;
    basis *b_ptr=&b;
    abgeleitet d;

    d.i = 1 + (b.i = 1);
    b_ptr->print_i ();  // "1 in basis"
    b_ptr = &d;
    b_ptr->print_i ();  // "2 in abgeleitet"
    // (waere "2 in basis", falls print_i nicht virtuell)
}
```

. . .Beispiel Virtuelle Funktion

Programmausgabe



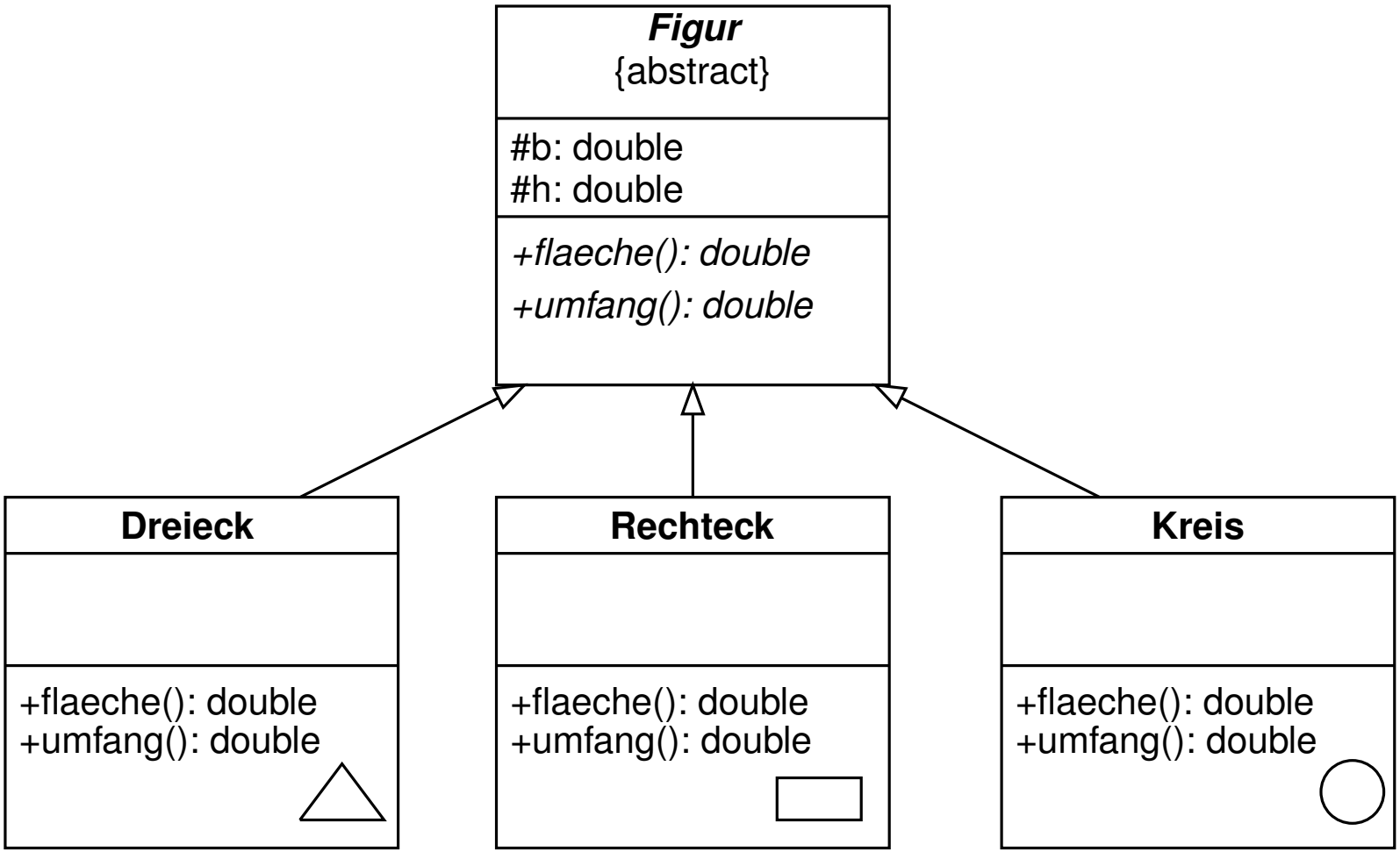
Abstrakte Klassen

virtual *datentyp funktionsbezeichner*(*parameterdeklarationen*) = 0;

- Wenn eine virtuelle Funktion der Basisklasse, die nicht in der abgeleiteten Klasse redefiniert ist, von einem Objekt der abgeleiteten Klasse aufgerufen wird, so wird die Funktion, so wie sie in der Basisklasse definiert ist, ausgeführt
- Oft gibt es in der Basisklasse keine sinnvolle Definition der virtuellen Funktion (z. B. Flächenberechnung geometrischer Objekte)
- Eine rein virtuelle Funktion besitzt in der Basisklasse keine Definition
- Eine Klasse, die mindestens eine rein virtuelle Funktion enthält, heißt abstrakte Klasse

Beispiel Geometrische Figuren

Abstrakte Klasse `Figur` mit abgeleiteten Klassen `Dreieck`, `Rechteck` und `Kreis`



Beispiel Geometrische Figuren figur.h...

```
#include <iostream>
#include <math.h>
// math.h enthaelt Dekl. mathematischer Funktionen

using namespace std;

class Figur {
protected:
  double b, h; // Dimensionen
public:
  void set_dim (double i, double j = 0)
  { b = i; h = j; }
  virtual double flaeche () = 0;
  virtual double umfang () = 0;
};
```

...Beispiel Geometrische Figuren figur.h...

```
class Dreieck: public Figur {
// b: Breite, h: Hoehe
public:
  double flaeche ()
  { return (b * 0.5 * h); }
  double umfang ()
  {
    return (sqrt ((b * 0.5) * (b * 0.5) + h * h)
            * 2 + b);
  }
};
```

...Beispiel Geometrische Figuren figur.h...

```
class Rechteck: public Figur {
// x: Breite, y: Hoehe
public:
  double flaeche ()
  {
    return (b * h);
  }
  double umfang ()
  {
    return (2 * b + 2 * h);
  }
};
```

...Beispiel Geometrische Figuren figur.h

```
class Kreis: public Figur {
// x: Durchmesser, y nicht benutzt
// PI in <math.h> definiert
public:
  double flaeche ()
  {
    return ((b * b / 4) * PI);
  }
  double umfang ()
  {
    return (b * PI);
  }
};
```

Beispiel Geometrische Figuren figurmain.cpp...

```
#include <iostream>
#include "figur.h"
using namespace std;
int const MENGE = 3;
int main ()
{
  double dim1, dim2, tot_flaeche;
  dreieck d; rechteck r; kreis k;
  figur *fig[MENGE];
  cout << "Dimensionen␣des␣Dreiecks␣(Figur␣1)␣eingeben:␣";
  cin >> dim1 >> dim2; d.set_dim (dim1, dim2);
  cout << "Dimensionen␣des␣Rechtecks␣(Figur␣2)␣eingeben:␣";
  cin >> dim1 >> dim2; r.set_dim (dim1, dim2);
  cout << "Dimension␣des␣Kreises␣(Figur␣3)␣eingeben:␣";
  cin >> dim1; k.set_dim (dim1);
  fig[0] = &d; fig[1] = &r; fig[2] = &k;
```

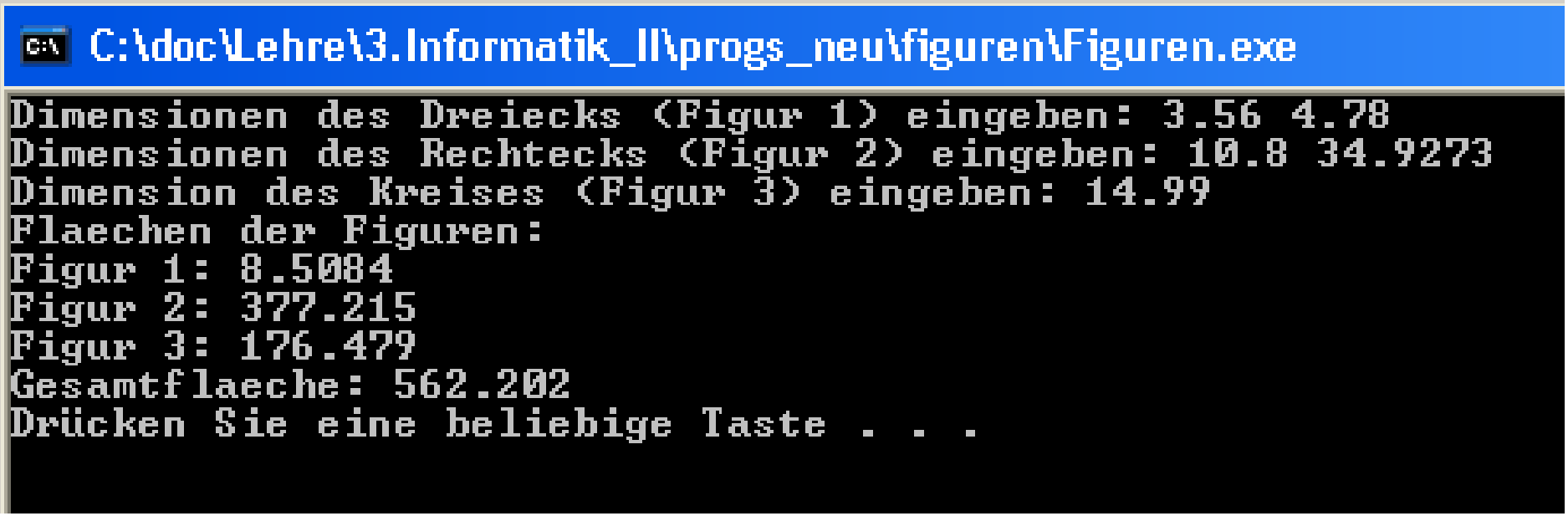
...Beispiel Geometrische Figuren figurmain.cpp

```
  cout << "Flaechen der Figuren:" << endl;
  for (int i = 0; i < MENGE; ++i) {
    cout << "Objekt " << i << ": ";
    cout << fig[i]->flaeche () << endl;
    tot_flaeche += fig[i]->flaeche ();
  }
  cout << "Gesamtflaeche: ";
  cout << tot_flaeche << endl;

  return (0);
}
```

Beispiel Geometrische Figuren: Benutzng

Programmausgabe



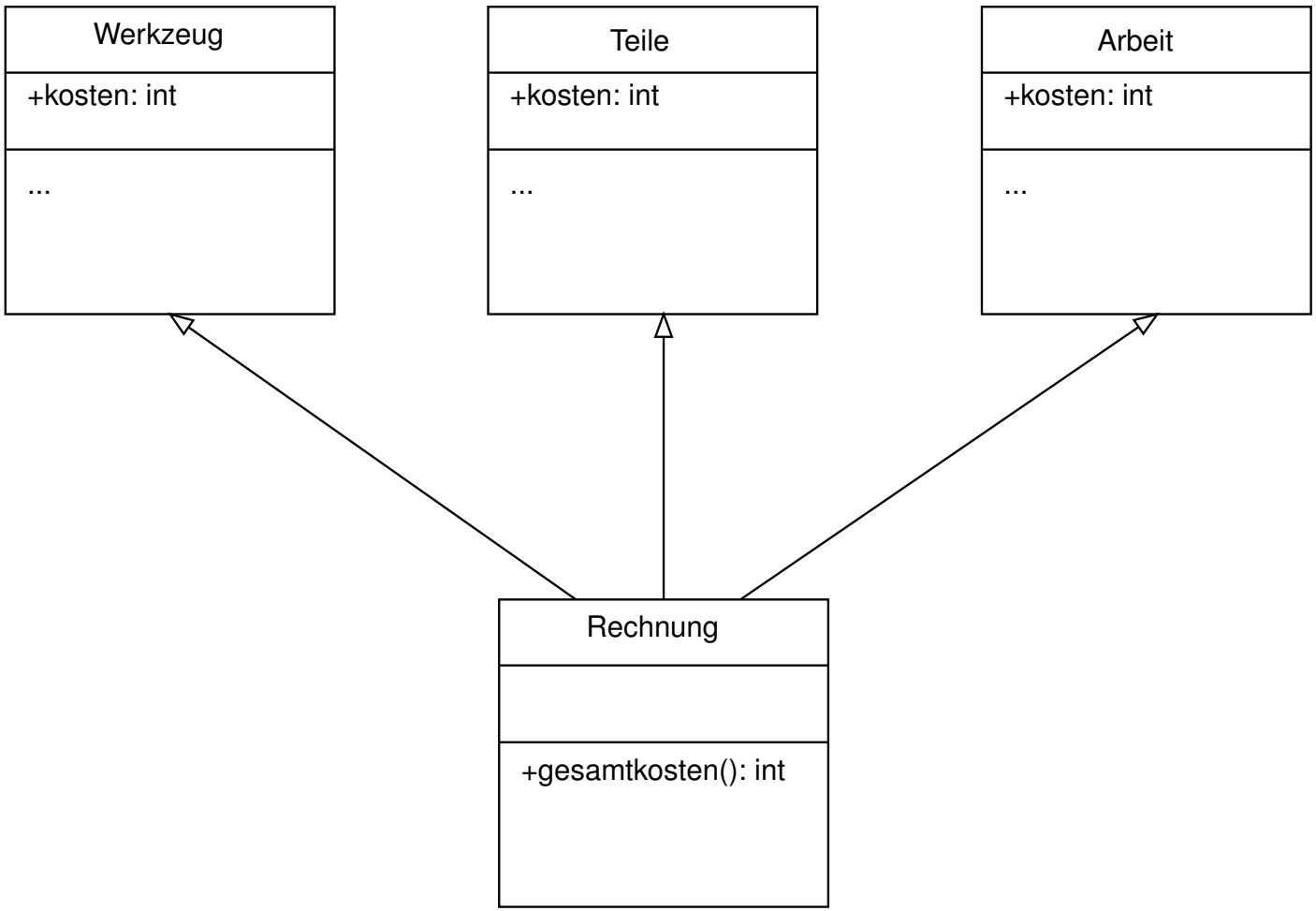
Virtuelle Klassen und Mehrfachvererbung

class *klassenbezeichner* : [*zugriff*] *basisklasse* [, [*zugriff*] *basisklasse*]* {
    daten und funktionen
};

- Eine Klasse kann von mehreren Klassen abgeleitet sein (Mehrfachvererbung, Multiple-inheritance)
- Bei Verwendung von Mehrfachvererbung können Mehrdeutigkeiten auftreten (Vererbung von Funktionen gleichen Namens und verschiedener Bedeutung aus verschiedenen Klassen)
- Es kann zu Duplizierung von Member-Variablen kommen

Beispiel Mehrdeutigkeit...

Rechnungspositionen `Werkzeug`, `Teile` und `Arbeit`



...Beispiel Mehrdeutigkeit

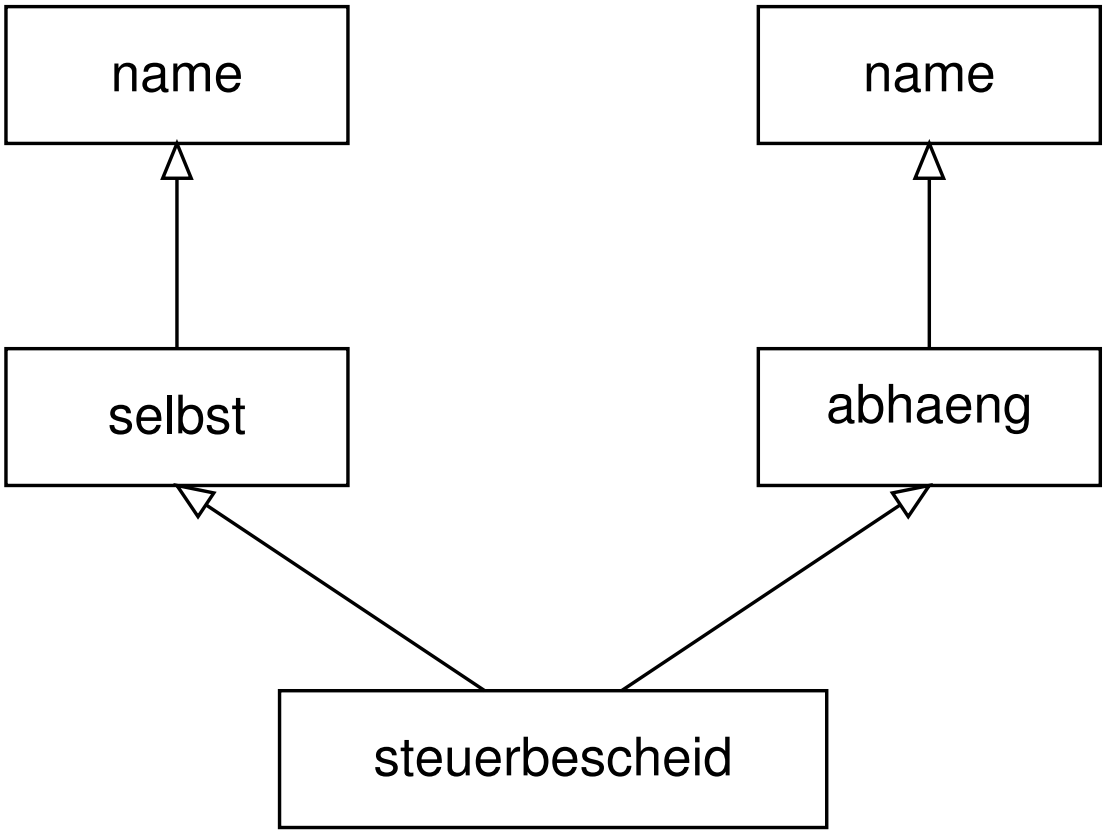
```
class Werkzeug {
public:
  int kosten ():
  // ...
};

class Teile {
public:
  int kosten ();
  // ...
};

class Arbeit {
public:
  int kosten ():
  // ...
};
```

```
class Rechnung : public Werkzeug,
  public Teile, public Arbeit {
public:
  int gesamtkosten ()
  {
  // Korrekt, eindeutig
    return(Werkzeug::kosten ()
      + Teile::kosten ()
      + Arbeit::kosten ());
  }
};
int fun ()
{
  int preis;
  Rechnung *ptr;
  // Inkorrekt, nicht eindeutig
  preis = ptr->kosten ();
}
```

Beispiel Duplizierung...




```
// name ist zweifach
steuerbescheid::selbst::name
steuerbescheid::abhaeng::name
```

...Beispiel Duplizierung

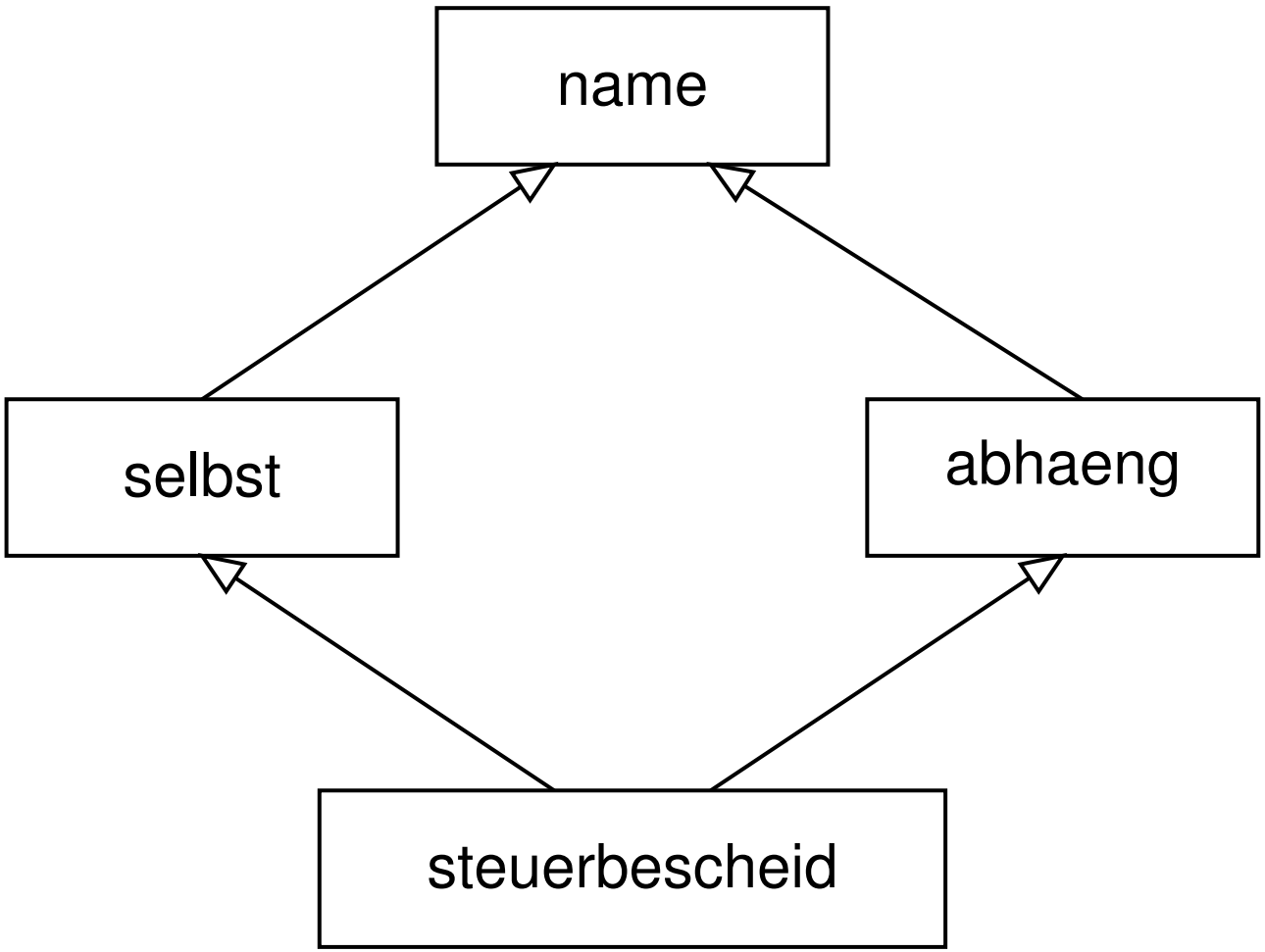
```
class name {
public:
  char *name;
  // ...
};
class selbst : public name {
public:
  int einnahmen;
  // ...
};
class abhaeng : public name {
public:
  int gehalt;
  // ...
};
class steuerbescheid : public selbst, public abhaeng {
  // name ist zweifach vorhanden!
};
```

Virtuelle Vererbung

class *klassenbezeichner* : virtual [*zugriff*] *basisklasse* {
    daten und funktionen
};

- Bei Mehrfachvererbung können zwei Basisklassen ihrerseits von einer gemeinsamen Basisklasse abgeleitet sein
- Wird eine Klasse von diesen beiden Basisklassen abgeleitet, so hat die abgeleitete Klasse zwei Member-Objekte der gemeinsamen Basis-Basisklasse
- Eine solche Duplizierung kann durch virtuelle Vererbung vermieden werden

Beispiel Virtuelle Vererbung...



...Beispiel Virtuelle Vererbung

```
class name {
public:
  char *name;
  // ...
};
class selbst : virtual public name {
public:
  int gehalt;
  // ...
};
class abhaeng : virtual public name {
public:
  int einnahmen;
  // ...
};
class steuerbescheid : public selbst,
                       public abhaeng {
  // name ist nur einfach vorhanden
};
```

Funktionsmaskierung...

- Ist in einer abgeleiteten Klasse eine angeforderte Funktion nicht definiert, so wird in der Basisklasse gesucht.
- Probleme treten bei Polymorphie auf.
- Sind in der Basisklasse zwei Funktionen gleichen Namens definiert und ist in der abgeleiteten Klasse nur eine dieser Funktionen redefiniert, so ist die andere Funktion der Basisklasse in der abgeleiteten Klasse nicht verfügbar.

. . .Funktionsmaskierung

```
class basis {
public:
  int fun (int i, int j) { return (i * j); }
  float fun (float f) { return (f * 2); }
};
class abgeleitet : public basis {
public:
  int fun (int i, int j) { return (i + j); }
};

int main () {
  abgeleitet test;
  int i;
  float f;
  i = test.fun (1, 3); // Legal, i = 4
  f = test.fun (4.0); // Illegal, in test nicht verfuegbar
  return (1);
}
```

Konstruktoren und Destruktoren

- Konstruktoren und Destruktoren verhalten sich in abgeleiteten Klassen anders als normale Member-Funktionen
- Bei Deklaration einer Instanz einer abgeleiteten Klasse wird erst der Konstruktor der Basisklasse und danach der Konstruktor der abgeleiteten Klasse ausgeführt
- Bei Zerstörung einer Instanz einer abgeleiteten Klasse wird erst der Destruktor der abgeleiteten Klasse und danach der Destruktor der Basisklasse ausgeführt
- Probleme gibt es bei Zeigern vom Typ der Basisklasse, die auf eine abgeleitete Klasse zeigen

Beispiel Konstruktoren und Destruktoren...

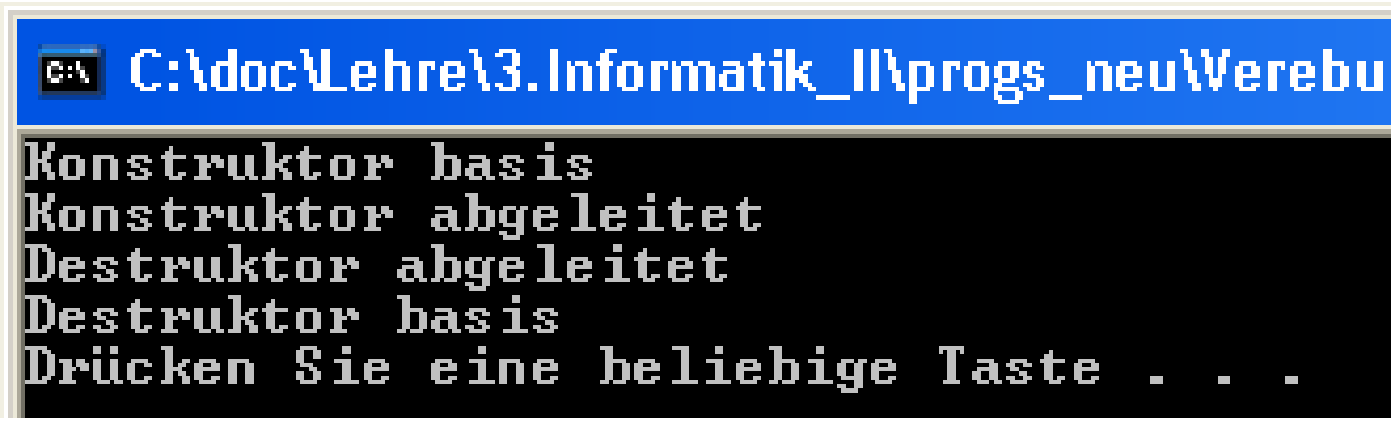
```
#include <iostream>
using namespace std;
class basis {
public:
  basis () { cout << "Konstruktor basis" << endl; }
  ~basis () { cout << "Destruktor basis"<< endl; }
};

class abgeleitet : public basis {
public:
  abgeleitet () { cout << "Konstruktor abgeleitet"<< endl; }
  ~abgeleitet () { cout << "Destruktor abgeleitet"<< endl; }
};
```

...Beispiel Konstruktoren und Destruktoren...

```
int main ()
{
  abgeleitet *test_ptr = new abgeleitet;
  delete test_ptr;
  test_ptr = 0; // zur Sicherheit
  return (0);
}
```

Ausgabe

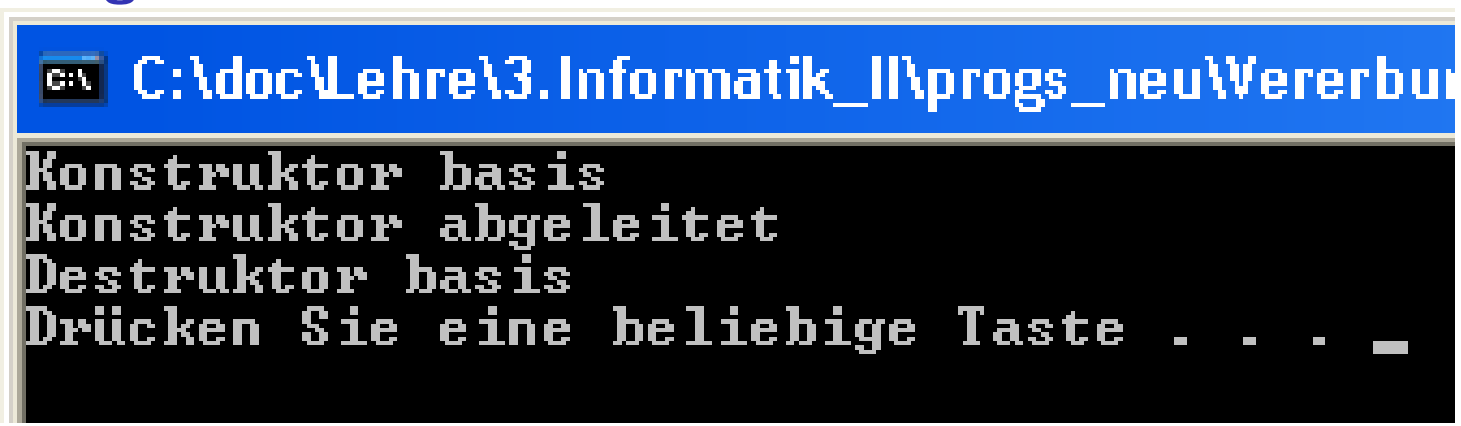


Virtuelle Destruktoren

```
// Legale Deklaration
basis *basis_ptr = new abgeleitet;

delete basis_ptr;
basis_ptr = 0;
```

Ausgabe

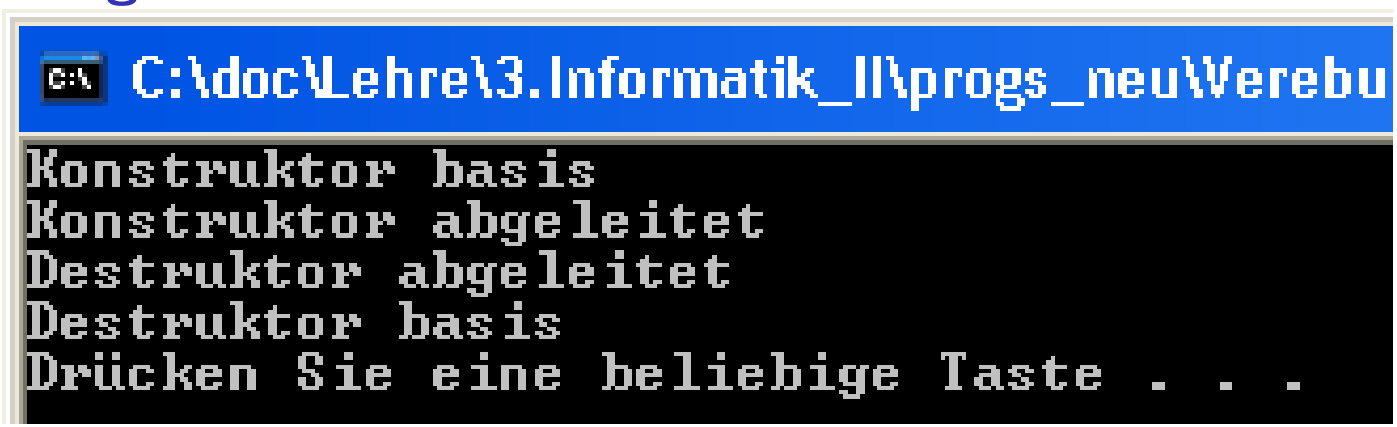


...Virtuelle Destruktoren

- Bei Zeigern auf Klassen wird die Funktion des Zeigertyps ausgeführt
- Ein virtueller Destruktor bewirkt die Ausführung des aktuellen Typs (Laufzeitbindung)

```
class basis {
public:
  basis () { cout << "Konstruktor basis"<< endl; }
  virtual ~basis () { cout << "Destruktor basis"<< endl; }
};
```

Ausgabe



Initialisierungsregeln

- Basisklassen werden in der Reihenfolge ihrer Deklaration initialisiert
- Members werden in der Reihenfolge ihrer Deklaration initialisiert
- Virtuelle Basisklassen werden vor jeder nicht-virtuellen Basisklasse erzeugt
- Die Reihenfolge erfolgt durch Tiefensuche links nach rechts im gerichteten Graphen der Vererbungshierarchie
- Destruktoren werden in umgekehrter Reihenfolge der Konstruktoren aufgerufen

Beispiel Initialisierung...

```
class werkzeug {
  // ...
public:
  werkzeug(char *w);
  ~werkzeug();
  // ...
};
class teile {
  // ...
public:
  teile(char *t);
  ~teile();
  // ...
};
```

```
class arbeit {
  // ...
public:
  arbeit(int a);
  ~arbeit();
  // ...
};
```

...Beispiel Initialisierung

```
class rechnung: public werkzeug, public teile,
                public arbeit {
  // ...
  spezial a; // Member-Klasse
             // mit Nicht-Standard-Konstruktor
public:
  rechnung(int r) : werkzeug("hammer"),
                    teile("schraube"),
                    arbeit(m),
                    a(m) { /* ... */}
  ~rechnung();
  // ...
};
```

Destruktorenreihenfolge

~a(), ~arbeit(), ~teile(),~werkzeug(), ~rechnung()

Konstruktionsschema...

```
class abstrakt_basis {
private:
  // meist leer, beschraenkt spaeteren Entwurf
protected:
  // statt private, ermoeglicht Vererbung
  // ...
public:
  // Standard-Konstruktor
  abstrakt_basis();
  // Copy-Konstruktor
  abstrakt_basis(const abstrakt_basis&);
  // Destruktor braucht Laufzeittyp
  virtual ~abstrakt_basis();
  // Ausdruck, rein virtuell
  virtual void print() = 0;
  // ...
};
```

...Konstruktionsschema

```
class abgeleitet :
  virtual public abstrakt_basis {
private:
  // ...
protected:
  // statt private fuer Vererbung
public:
  // Schnittstelle realisiert die Instanz
  // Standard-Konstruktor
  abgeleitet();
  // Copy-Konstruktor
  abgeleitet(const abgeleitet&);
  // Zuweisung
  abgeleitet &operator =(const abgeleitet&);
  // Konkretes Ausdrucken
  void print();
  // ...
};
```

Wichtige Eigenschaften

- Eine virtuelle Funktion und ihre abgeleiteten Instanzen müssen die selbe Parameterstruktur und den selben Datentyp des Rückgabewertes haben
- Nicht-virtuelle Member-Funktionen können die selbe Parameterstruktur haben und verschiedenen Rückgabetyp
- Alle Member-Funktionen mit Ausnahme von Konstruktoren und redefinierten `new` und `delete` können virtuell sein
- Konstruktoren, Destruktoren, Zuweisungsoperator (=) und `friend`s werden nicht vererbt
- Zugriff auf Members kann bei Vererbung nur eingeschränkt, nicht erweitert werden

Vererbung Teil 2

- Mit virtuellen Funktionen wird die Laufzeitbindung realisiert.
- Mit virtuellen Funktionen werden in C++ abstrakte Klassen implementiert.
- Bei Mehrfachvererbung kann es zu Mehrdeutigkeiten und Duplizierung von Member-Variablen kommen.
- Die Duplizierung von Member-Variablen kann durch virtuelle Vererbung vermieden werden.
- Konstruktoren und Destruktoren verhalten sich in abgeleiteten Klassen anders als normale Member-Funktionen.