Software-Technik: Vom Programmierer zur erfolgreichen …

1. Von der Idee zur Software
2. Funktionen und Datenstr
3. Organisation des Quellc
4. Werte- und Referenzsem
5. Entwurf von Algorithmen
6. Fehlersuche und –behandlung
7. Software-Entwicklung im Team
8. Abstrakte Datentypen: Einheit von Daten und Funktionalität
9. Vielgestaltigkeit (Polymorphie)
10. Entwurfsprinzipien für Software

Anhang A: Die Familie der C-Sprachen

Anhang B: Grundlagen der C++ und der Java-Programmierung

Lehrbuch: 4.3
Kompendium, 3. Auflage: 8.10, 9.3
Kompendium, 4. Auflage: 9.2

Software-Technik: Vom Programmierer zur erfolgreichen …

Folien mit gelben Punkten ● am oberen rechten Rand sind weniger wichtiger für das Verständnis der nachfolgenden Kapitel.

● Folien seit erster Vorstellung in Vorlesung (10.5.07) angepasst

Prinzip des dynamischen Bindens an einem hypothetischen Beispiel aus C++-Sicht

Virtuelle Methoden

```
class Mitarbeiter {
   public:
      float Gehalt() const { return 2500; }
      int   Unterg() const { return 0; }
};

class Chef: public Mitarbeiter {
   public:
      float Gehalt() const { return 40000; }
      int   Unterg() const { return 20; }
};

void PrintPerson (const Mitarbeiter* pers) {
   cout<<pers->Gehalt()<<"  "<<pers->Unterg();
}
. . .

Chef* schroeder = new Chef; Mitarbeiter* mueller = new Mitarbeiter;
PrintPerson(mueller); cout << endl; PrintPerson(schroeder);
```

Welche Methode wird jeweils aufgerufen?

- Chef::GetGehalt oder
- Mitarbeiter::GetGehalt?

Ausgabe ist?

1. 2500 0 2500 0
2. 40000 20 40000 20
3. 21250 10 21250 10
4. 2500 0 40000 20

Virtuelle Methoden

```
class Mitarbeiter {
   public:
      float Gehalt() const { return 2500; }
      int   Unterg() const { return 0; }
};

class Chef: public Mitarbeiter {
   public:
      float Gehalt() const { return 40000; }
      int   Unterg() const { return 20; }
};

void PrintPerson (const Mitarbeiter* pers) {
   cout<<pers->Gehalt()<<"  "<<pers->Unterg();
}
. . .

Chef* schroeder = new Chef; Mitarbeiter* mueller = new Mitarbeiter;
PrintPerson(mueller); cout << endl; PrintPerson(schroeder);
```

Welche Methode wird jeweils aufgerufen?

- Chef::GetGehalt oder
- Mitarbeiter::GetGehalt?

Ausgabe ist?

1. 2500 0 2500 0
2. 40000 20 40000 20
3. 2
4. 2

Die Ausgabe ist:

2500 0

2500 0

Das ist offensichtlich nicht das Gewünschte!

Virtuelle Methoden

```
void PrintPerson (const Mitarbeiter* pers) {
   cout<<pers->Gehalt()<<"  "<<
                    pers->Unterg();
}
. . .

Chef* schroeder = new Chef;
Mitarbeiter* mueller = new Mitarbeiter;
PrintPerson(mueller); cout << endl;
    PrintPerson(schroeder);
```

Welche Methode wird jeweils aufgerufen?

- Chef::GetGehalt oder
- Mitarbeiter::GetGehalt?

Ausgabe ist?

1. 2500 0 2500 0
2. 40000 20 40000 20
3. 21250 10 21250 10
4. 2500 0 40000 20

Die Ausgabe ist:
2500 0
2500 0

Das ist offensichtlich nicht das Gewünschte!

Virtuelle Methoden (3)

```
class Mitarbeiter {
   public:
      virtual float Gehalt() const { return 2500; }
      virtual int    Unterg() const { return 0; }
};

class Chef: public Mitarbeiter {
   public:
      virtual float Gehalt() const { return 40000; }
      virtual int    Unterg() const { return 20; }
};

void PrintPerson (const Mitarbeiter* pers) {
   cout<<pers->Gehalt()<<"  "<<pers->Unterg();
}
. . .

Mitarbeiter* schroeder = new Chef; Mitarbeiter* mueller = new Mitarbeiter;
PrintPerson(mueller); cout << endl; PrintPerson(schroeder);
```

Jetzt werden die Methoden gemäß dem dynamischen Typ -- d.h. gemäß dem Typ des Objektes, an das der Zeiger gebunden ist -- selektiert !

Die Ausgabe ist also:
2500 0
40000 20

Virtuelle Methoden

```
class Mitarbeiter {
   public:
      virtual float Gehalt() const { return 2500; }
      virtual int   Unterg() const { return 0; }
};

class Chef: public Mitarbeiter {
   public:
      virtual float Gehalt() const { return 40000; }
      virtual int   Unterg() const { return 20; }
};

void PrintPerson (const Mitarbeiter& pers) {
   cout<<pers.Gehalt()<<"  "<<pers.Unterg();
}
. . .

Chef* schroeder = new Chef; Mitarbeiter* mueller = new Mitarbeiter;
PrintPerson(*mueller); cout << endl; PrintPerson(*schroeder);
```

Welche Methode wird jeweils aufgerufen?

- Chef::GetGehalt oder
- Mitarbeiter::GetGehalt?

Ausgabe ist?

1. 2500 0 2500 0
2. 2500 0 40000 20
3. 21250 10 21250 10
4. 40000 0 40000 10

Virtuelle Methoden (4)

```
class Mitarbeiter { ... wie bei (3) ... };
class Chef: public Mitarbeiter {... wie bei (3) ... };
void PrintPerson1 (const Mitarbeiter* pers) {
   cout<<pers->Gehalt()<<"  "<<pers->Unterg();
}
void PrintPerson2 (const Mitarbeiter& pers) {
   cout<<pers.Gehalt()<<"  "<<pers.Unterg();
}
void PrintPerson3 (Mitarbeiter pers) {
   cout<<pers.Gehalt()<<"  "<<pers.Unterg();
}
```

Dynamisches Binden funktioniert sowohl bei Bindung über Zeiger als auch bei Bindung über Referenzen, d.h. hier bei den Methoden

- *PrintPerson1* und
- *PrintPerson2.*

Dynamisches Binden funktioniert nicht beim direkten Zugriff, d.h. hier bei der Methode

- *PrintPerson3.*

Virtuelle Methoden Motivation von „override“

```
class Mitarbeiter {
   public:
      virtual float Gehalt() const { return 2500; }
      virtual int    Unterg() const { return 0; }
};

class Chef: public Mitarbeiter {
   public:
      virtual float Gehalt() const { return 40000; }
      virtual int    Unterg()          { return 20; }
};

void PrintPerson (const Mitarbeiter& pers) {
   cout<<pers.Gehalt()<<"  "<<pers.Unterg();
}
. . .

Chef* schroeder = new Chef; Mitarbeiter* mueller = new Mitarbeiter;
PrintPerson(*mueller); cout << endl; PrintPerson(*schroeder);
```

Welche Methode wird jeweils aufgerufen?

- Chef::GetGehalt oder
- Mitarbeiter::GetGehalt?

Ausgabe ist?

1. 2500 0 2500 0
2. 2500 0 40000 20
3. 2500 0 40000 0
4. 2500 0 2500 20

Virtuelle Methoden ohne „override“

```
class Mitarbeiter {
   public:
      virtual float Gehalt() const { return 2500; }
      virtual int    Unterg() const { return 0; }
};

class Chef: public Mitarbeiter {
   public:
      virtual float Gehalt() const { return 40000; }
      virtual int    Unterg()          { return 20; }
};
```

Welche Methode wird jeweils aufgerufen?

- Chef::GetGehalt oder
- Mitarbeiter::GetGehalt?

Ausgabe ist?

1. 2500 0 2500 0
2. 2500 0 40000 20
3. 2500 0 40000 0
4. 2500 0 2500 20

Dynamisches Binden unterbliebt für Unterg, da const fehlt. Das ist eine ganz andere Methode.

Virtuelle Methoden Schlüsselwort „override“

```
class Mitarbeiter {
   public:
      virtual float Gehalt() const { return 2500; }
      virtual int    Unterg() const { return 0; }
};

class Chef: public Mitarbeiter {
   public:
      virtual float Gehalt() const override { return 40000; }
      virtual int    Unterg()       override    { return 20; }
};
```

Das würde einen Compilerfehler auslösen!!!
Also override (seit C++-11) unbedingt verwenden.

Jetzt die Theorie

Polymorphie

Eine Schnittstelle (abstrakter Datentyp) kann mehrere, verschiedene konkrete Datenstrukturen abstrahieren.

Wird die Entscheidung, welche Funktion konkret die Schnittstellenfunktion implementiert, zur Compilezeit festgelegt, spricht man von **statischer Bindung.**

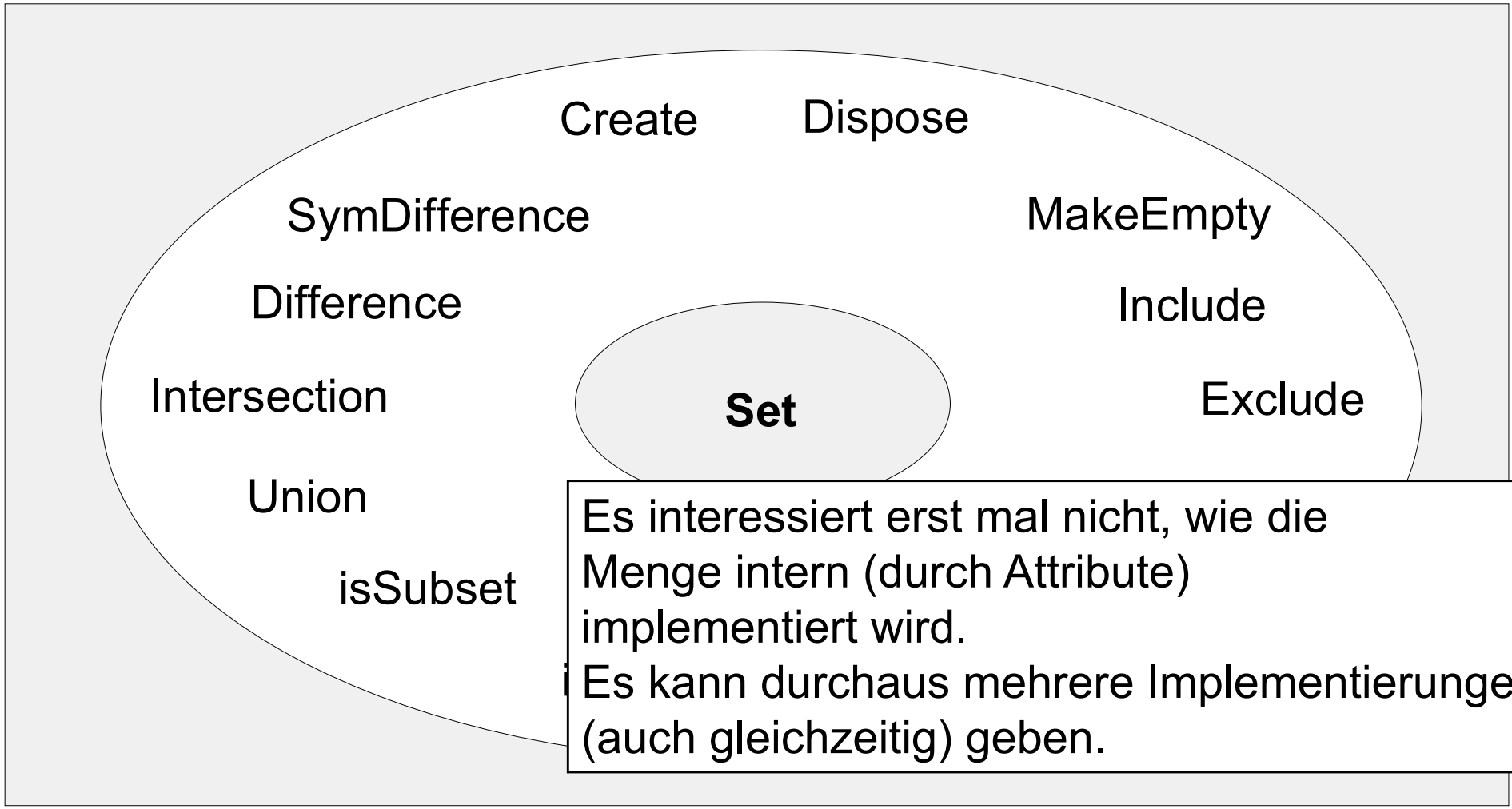
Erfolgt diese Entscheidung zur Laufzeit, ist eine parallele Nutzung verschiedener Implementierungen möglich und man spricht von **dynamischer Bindung**.

Dabei obliegt die Verantwortung für die Auswahl der richtigen Funktion dem Ersteller des Datentyps und nicht dem Nutzer.

In diesem Fall kann eine Variable (von einen bestimmten (abstrakten) Datentyp) in verschiedener Gestalt vorkommen.

Man spricht von Vielgestaltigkeit bzw. **Polymorphie.**

Menge als Abstrakter Datentyp



Es interessiert erst mal nicht, wie die Menge intern (durch Attribute) implementiert wird.
Es kann durchaus mehrere Implementierungen (auch gleichzeitig) geben.

Automatische Polymorphie (2)

```
class Vorgang {
public:
  virtual double getDauer() = 0;
};
```

C++

```
interface Vorgang {
    double getDauer();
}
```

Java

Der Vorgang mit den Funktionszeigern und den Umleitungs-Schnittstellenfunktionen (siehe Buch, Kap. 9) wird durch eine Klasse oder in Java (wahlweise) durch ein Interface ersetzt, die nichts weiter als die Funktionsdeklarationen der Schnittstellenfunktionen enthält, aber keine Implementationen.

Die administrativen Datenstrukturen (Funktionszeiger) und Umleitungsfunktionen werden praktisch vom Compiler erzeugt.

Automatische Polymorphie (3)

```
class Transportvorgang :
                    public Vorgang {
  /* …*/
  virtual double getDauer();
};
```

C++

```
class Transportvorgang
              implements Vorgang {
  /* ...*/
  double getDauer() { return ...; }
}
```

Java

Die konkreten Vorgänge (abgeleitete Klassen von Vorgang) werden anschließend in eigenen Klassen implementiert.

Gibt es in Transportvorgang dann Funktionen mit identischen Rückgabewerten, Namen und Argumenten wie die Schnittstellenfunktion, so ordnet der Compiler diese Funktionen den Schnittstellenfunktionen zu (in unserem Modell entspricht das der Übernahme der Funktionszeiger).

Zusammenfassung (2)

```
class Vorgang {
public:
 int getId() {….};
 virtual double getDauer() = 0;
 virtual double getFruehAnf() {..};
};
```

Und genau aus diesen Gründen (und sonst keinen) sollte eine Methode als

- virtuell (virtual),
- rein virtuell oder
- nicht virtuell

deklariert werden.

```
class Vorgang {
public:
 // kann nicht überschrieben werden
 int getId() {….};

 // muss überschrieben werden
 virtual double getDauer() = 0;

 // kann überschrieben werden
 virtual double getFruehAnf() {..};

  /*Abstrakte Methode können sogar
  implementiert werden und bieten
  Erben Default-Implementieung zur
  Nutzung an */
  virtual int meth() = 0 {…….} ;
};
```

Automatische Polymorphie (4)

Das vorherige Beispiel lässt sich somit in objektorientierten Sprachen viel einfacher umsetzen.

Dabei ist für öffentliche Methoden der Schnittstelle, die dynamisch gebunden werden sollen, in C++ das Schlüsselwort **virtual** voranzustellen -- in Java werden immer **alle** Methoden dynamisch gebunden.

Eine Klasse heißt **abstrakte Klasse,** wenn die Implementierung einzelner oder aller virtuellen Methoden fehlt, was in C++ auch direkt durch den Zusatz „ = 0“ hinter der Funktionsdeklaration offenbar wird.

Eine abstrakte Klasse ist unvollständig, weil die abstrakten Methoden in ihr nicht aufgerufen werden können, es gibt für sie noch keine zugeordnete Implementierung (Funktionzeiger ist noch NULL).

Daher wird die Instanziierung (d.h. der Aufruf des Konstruktors) einer abstrakten Klasse auch vom Compiler mit einer Fehlermeldung quittiert.

Virtuelle Methoden (2)

In beiden Fällen werden die Methoden der Basisklasse aufgerufen, da bereits zur Übersetzungszeit auf Grund des Zeigertyps ***Mitarbeiter* pers*** die Methode gemäß dem **statischen Typ *Mitarbeiter*** ausgewählt wird **(statische Bindung).**

Wird jedoch in der Basisklasse der Deklaration der Methoden *Gehalt* und *Unterg* das Schlüsselwort *virtual* vorangestellt, so wird erst zur Laufzeit aufgrund der Klasse des aktuellen Objektes entschieden, welche Methode wirklich abgearbeitet wird **(dynamische Bindung),** beim zweiten Aufruf also *Chef::Gehalt* und *Chef::Unterg*.

Virtuelle Methoden (5)

Die Klasse *Mitarbeiter* ist *polymorph*, weil Zeiger vom Typ *Mitarbeiter** sowohl an Objekte vom Basistyp Mitarbeiter als auch an Objekte vom abgeleiteten Typ Chef gebunden werden können.

Eine Klasse ist *polymorph,* wenn sie mindestens eine Methode besitzt (geerbt oder selbst deklariert), die virtuell ist, sodass diese dann in einer abgeleiteten Klasse redefiniert (überschrieben) werden kann.

Wird die Methode aus der Basisklasse oder aus abgeleiteten Klasse aufgerufen?

Wir unterscheiden bei jeder Variablen zwischen dem dynamischen Typ und dem statischen Typ.

Beispiel

```
Obst*   ob1 = new Apfel();
```

Der statische Typ von ob1 ist immer Obst (Zeiger auf Obst).

Der dynamische Typ von ob1 kann sich ändern und ist zur Zeit Apfel (Zeiger auf Apfel ganz genau).

Wird die Methode aus der Basisklasse oder aus abgeleiteten Klasse aufgerufen?

Liegt ein Wert (also keine Referenz und kein Zeiger) vor, ist der statische Typ immer gleich dem dynamischen Typ. (Merksatz M1)

Nur bei Referenz oder Zeigern können der dynamische und statische Typ sich unterscheiden (d.h. der Typ der Instanz, auf die verwiesen wird). (Merksatz M2)

Bei virtuellen Methoden entscheidet der dynamische Typ über die Methode, die aufgerufen wird. (Merksatz M3)

Bei nicht virtuellen Methoden entscheidet der statische Typ über die Methode, die aufgerufen wird. (Merksatz M4)

Clicker

```
void funk() {
Vorgang* vv[3] ;
vv[0] = new TransVorg ();
vv[1] = new ProdVorg ();
vv[2] = new TransVorg();

for (int i=0;i<3;++i) {
        vv[i]->print();
} }
```

```
class Vorgang {  public:
   virtual void print() const {cout << "V"; }
}
class TransVorg : public Vorgang { public:
   virtual void print() const {cout << "T"; }
}
class ProdVorg : public Vorgang { public:
   virtual void print() const {cout << "P"; }
}
```

Wie ist die Ausgabe auf dem Bildschirm nach Ausführung von funk ?

1. VVV 2. TPT 3. VTP 4. TTT

Clicker

```
void funk() {
Vorgang* vv[3] ;
vv[0] = new TransVorg ();
vv[1] = new ProdVorg ();
vv[2] = new TransVorg();

for (int i=0;i<3;++i) {
        vv[i]->print();
} }
```

```
class Vorgang {  public:
   virtual void print() const {cout << "V"; }
}
class TransVorg : public Vorgang { public:
   virtual void print() const {cout << "T"; }
}
class ProdVorg : public Vorgang { public:
   virtual void print() const {cout << "P"; }
}
```

Wie ist die Ausgabe auf dem Bildschirm nach Ausführung von funk ?

1. VVV 2. TPT 3. VTP 4. TTT

Bei virtuellen Methoden entscheidet der dynamische Typ über die Methode, die aufgerufen wird. (Merksatz M3)

Ergebnis:
___ 1 __ 2 __ 3 __4 (wg. M2 und M3)

Clicker

```
void funk() {
Vorgang* vv[3];
vv[0] = new TransVorg ();
vv[1] = new ProdVorg ();
vv[2] = new TransVorg ();

for (int i=0;i<3;++i) {
        vv[i]->print();
} }
```

```
class Vorgang {  public:
    void print() const {cout << "V"; }
}
class TransVorg : public Vorgang { public:
    void print() const {cout << "T"; }
}
class ProdVorg : public Vorgang { public:
    void print()const {cout << "P"; }
}
```

Wie ist die Ausgabe auf dem Bildschirm nach Ausführung von funk ?
1. VVV 2. TPT 3. VTP 4. TTT

Clicker

```
void funk() {
Vorgang* vv[3];
vv[0] = new TransVorg ();
vv[1] = new ProdVorg ();
vv[2] = new TransVorg ();

for (int i=0;i<3;++i) {
        vv[i]->print();
} }
```

```
class Vorgang {  public:
    void print() const {cout << "V"; }
}
class TransVorg : public Vorgang { public:
    void print() const {cout << "T"; }
}
class ProdVorg : public Vorgang { public:
    void print()const {cout << "P"; }
```

Wie ist die Ausgabe auf dem Bildschirm nach Ausführung von funk ?
1. VVV 2. TPT 3. VTP 4. TTT

Bei nicht virtuellen Methoden entscheidet der statische Typ über die Methode, die aufgerufen wird. (Merksatz M4)

Ergebnis:
__1 __2 __3 __4 (wg. M2 und M4)

Clicker

```
void funk() {
Vorgang  v;
TransVorg  tv;
ProdVorg pv;

Vorgang v[3];
  v[0] = v;    v[1]=tv;    v[2]=pv;
for (int i=0;  i<3;  ++i) {
          v[i].print();
} }
```

```
class Vorgang {  public:
    virtual void print() const {cout << "V"; }
}
class TransVorg : public Vorgang { public:
    virtual void print() const {cout << "T"; }
}
class ProdVorg : public Vorgang { public:
    virtual void print() const {cout << "P"; }
}
```

Wie ist die Ausgabe auf dem Bildschirm nach Ausführung von funk ?

1. VVV 2. TPT 3. VTP 4. TTT

Clicker

```
void funk() {
Vorgang  v;
TransVorg  tv;
ProdVorg pv;

Vorgang v[3];
  v[0] = v;    v[1]=tv;    v[2]=pv;
for (int i=0;  i<3;  ++i) {
          v[i].print();
} }
```

```
class Vorgang {  public:
    virtual void print() const {cout << "V"; }
}
class TransVorg : public Vorgang { public:
    virtual void print() const {cout << "T"; }
}
class ProdVorg : public Vorgang { public:
    virtual void print() const {cout << "P"; }
}
```

Wie ist die Ausgabe auf dem Bildschirm nach Ausführung von funk ?

1. VVV 2. TPT 3. VTP 4. TTT

Liegt ein Wert (also keine Referenz und kein Zeiger) vor, ist der statische Typ immer gleich dem dynamischen Typ. (Merksatz M1)

Ergebnis:

__ 1 __ 2 __ 3 __ 4

Clicker

```
void funk() {
Vorgang** vv = new Vorgang*[3];
vv[0] = new TransVorg ();
vv[1] = new ProdVorg ();
vv[2] = new TransVorg();

for (int i=0;i<3;++i) {
        vv[i]->print();
} }
```

```
class Vorgang {  public:
   virtual void print() const {cout << "V"; }
}
class TransVorg : public Vorgang { public:
   virtual void print() const {cout << "T"; }
}
class ProdVorg : public Vorgang { public:
   virtual void print() const {cout << "P"; }
}
```

Wie ist die Ausgabe auf dem Bildschirm nach Ausführung von funk ?

1. VVV 2. TPT 3. VTP 4. TTT

Clicker

```
void funk() {
Vorgang** vv = new Vorgang*[3];
vv[0] = new TransVorg ();
vv[1] = new ProdVorg ();
vv[2] = new TransVorg();

for (int i=0;i<3;++i) {
        vv[i]->print();
} }
```

```
class Vorgang {  public:
   virtual void print() const {cout << "V"; }
}
class TransVorg : public Vorgang { public:
   virtual void print() const {cout << "T"; }
}
class ProdVorg : public Vorgang { public:
   virtual void print() const {cout << "P"; }
}
```

Wie ist die Ausgabe auf dem Bildschirm nach Ausführung von funk ?

1. VVV 2. TPT 3. VTP 4. TTT

Ergebnis:
___ 1 _ 2 __ 3 __4

Virtuelle Destruktoren

```
class A {
    int* px;
  public:
    A()  {cout<<"A+"; px = new int;}
    virtual
    ~A() {cout<<"A-"; delete px;}
    // . . .
};
class B : public A {
    int* py;
  public:
    B()  {cout<<"B+"; py = new int;}
    virtual
    ~B() {cout<<"B-"; delete py;}
    // . . .
};
```

```
void f() {
  A* pA = new B;
  // . . .
  delete pA;
}
int main() {
  f();
  // Speicherleck, wenn Destruktor
  // in Basisklasse nicht virtuell !
  // . . .
}
```

Ausgabe mit virt. Destr.: A+B+B-A-
Ausgabe ohne virt. Destr.: A+B+A-

Virtuelle Destruktoren

```
class A {
    int* px;
  public:
    A(int w)  {cout<<"A+";
                    px = new int(w);}
    virtual
    ~A() {cout<<"A-"; delete px;}
    // . . .
};

class B : public A {
    int* py;
  public:
    B(int aw, int bw) : A(aw) {
     cout<<"B+"; py = new int(bw);}
    virtual
    ~B() {cout<<"B-"; delete py;}
    // . . .
};
```

```
void f() {
  A  a(17);
  B b(16,  33);
}

int main() {
  f();
  // Speicherleck, wenn Destruktor
  // in Basisklasse nicht virtuell !
  // . . .
}
```

Werte- und Referenzsemantik in Zusammenhang mit Polymorphie

Immer wenn Variablen in Wertesemantik vorliegen, ist ihr Typ genau bekannt und Funktionsaufrufe können statisch gebunden werden. Wenn verschiedene Realisierungen des Datentyps unterschiedlich groß sein können und wir im Sinne eines polymorphen Datentyps nicht wissen, welche Gestalt ein Vorgang gerade annimmt, ist das Anlegen einer polymorphen Variable in Wertesemantik schlicht unmöglich. Eine Vorbedingung für das Auftreten von dynamischer Bindung ist also, dass wir es mit Zeigern oder Referenzen zu tun haben. Da in C++ der Anwender entscheiden soll, ob er gerade Polymorphie nutzen will oder nicht, gesteht C++ dem Entwickler das Recht zu, konkrete Objekte sowohl in Werte- als auch in (polymorphiefähiger) Zeiger/Referenz-Semantik zu verwenden. In Java werden alle Funktionsaufrufe dynamisch gebunden, entsprechend sind alle Variablen in Java -- bis auf die elementaren Datentypen -- Referenzen auf Objekte (im Heap).

Werte- und Referenzsemantik in Zusammenhang mit Polymorphie (2)

Schreibt der C++-Anwender vector<TransportVorgang>, so drückt er damit aus, dass er Wert-Instanzen von (ausschließlich) Transportvorgängen speichern möchte (bestehend aus Anzahl, Weglänge und Einheitsdauer). Der Typ ist damit eindeutig festgelegt und Polymorphie ist unnötig (und unmöglich).

Schreibt er hingegen vector<TransportVorgang*> so drückt er damit aus, dass er Referenzen auf Transportvorgänge speichern will, wobei es dann durchaus möglich ist, dass es mehrere, verschiedene (Unterklassen von) Transportvorgänge geben kann, von denen automatisch die richtige Funktion aufgerufen wird.

Diese Überlegungen braucht ein Java-Entwickler nicht anzustellen; semantisch entspricht bei ihm Vector<Vorgang> immer der C++-Variante vector<Vorgang&> (oder vector<Vorgang*>).

Rein virtuelle Methoden und abstrakte Klassen

```
class Zahl {  // abstrakte Klasse
   public:
      virtual void Print() const = 0;
};

class Integer : public Zahl {
      . . .
      virtual void Print() const override   { ... }
};

class Float : public Zahl {
      . . .
      virtual void Print() const override { ... }
};
```

```
class Complex : public Zahl {
      . . .
      virtual void Print() const {
... }
};
 int main() {
   Zahl* vec[10];
   vec[0] = new
Integer(1234);
   . . .
   for(int i=0; i<3; i++)
         vec[i]->Print();
}
```

Rein virtuelle Funktionen werden durch das „= 0“ gekennzeichnet, sie dienen nur der Spezifikation. Eine Klasse mit mindestens einer rein virtuellen Funktion ist abstrakt und kann nicht instanziiert werden!

Zusammenfassung

Die dynamische Bindung kann nur dort erfolgen, wo über den Punktoperator (oder Pfeiloperator) eine Methode aufgerufen wird, die mit identischem Prototyp mehrfach in verschiedenen Implementierungen realisiert wurde (Überschreiben des Funktionszeigers).