Objektorientierte Programmierung

Klasse: Gruppe, Einheit (Menge von Objekten) mit gemeinsamen, sich von anderen unterscheidenden Merkmalen (siehe Duden).

Klassifikation, Klassifizierung: Das Einordnen, die Einteilung [in aufgestellte Klassen] (siehe Duden).

Objektorientierte Programmierung

Simulation der realen Welt durch Abbildung in Rechner.

- **Objekte** der „realen Welt“ werden in die „virtuelle Welt“ eingebettet.
- **Objekte im Sinne des OOP**
  - sind Abstraktionen von Dingen (Objekten) der realen Welt.
  - bestehen aus Daten = Attribute, Eigenschaften
    Operationen = Methoden, Funktionen
- Objekte mit den gleichen Eigenschaften (Daten) und Operationen werden zu **Klassen** zusammengefasst.

Wiederverwendbarkeit

• von getesteten und optimierten Klassen!!!

Beim Entwurf einer Klasse muss man zunächst die folgenden Fragen beantworten:

– Welche gemeinsamen Eigenschaften (Attribute) besitzen die Objekte der Klasse?

– Welche gemeinsamen Verhaltensweisen (Methoden) zeigen die Objekte der Klassen?

– Mit welchen Methoden möchte man auf den Objekten der Klasse operieren?

Abstrakte Datentypen

– Beispiel: Beschreibung einer Klasse „Point2D“:

• Attribute	xCoordinate:	int
	yCoordinate:	int
	color:	short (enum?)
• Operationen	x( )	x-Koordinate zurückgeben
	y( )	y-Koordinate zurückgeben
	setCoordinates(int, int)	Koordinaten ändern
	getColor( )	Farbwert zurückgeben
	setColor(short)	Farbwert setzen

– Ein **abstrakter Datentyp (ADT)** fasst
  - die Daten (Eigenschaften) und
  - die Methoden (Funktionen), mit denen die Datentypen bearbeitet werden dürfen, zusammen.

Eine erste Klasse

– Eine Klasse ist ein abstrakter Datentyp, der in einer Programmiersprache (C++, Java, usw.) formuliert ist.

```
class Point2D { // nach Schlüsselwort „class“ folgt der Klassenname
  public:
        int     x( );
        int     y( );
        void    setCoordinates(int, int);
        short   getColor( );
        void    setColor(short);
  private:
        int     xCoordinate;
        int     yCoordinate;
        short   color;
};
// Man beachte das Semikolon am Ende der Definition!
```

Eine erste Klasse

- Eine C++ Klasse hat folgende typische Gestalt:

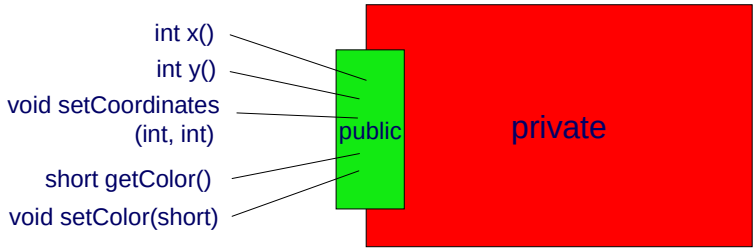
```
class Klassenname {
   public:
     Rückgabetyp Elementfunktion1(Parameterliste1);
     Rückgabetyp Elementfunktion2(Parameterliste2);
     // ... weitere Elementfunktionen
   private:
     Datentyp Attribut1;
     Datentyp Attribut2;
     //  .... weitere Attribute
};
```

- Wir werden die Details der Syntax einer Klassendefinition nach und nach kennen lernen.

Eine erste Klasse

– Die Methoden (und Attribute) in dem mit „public“ gekennzeichneten Teil der Deklaration bilden die öffentliche Schnittstelle der Klasse.

– public = öffentliche Schnittstelle = von jedem benutzbare Operationen



– Die Attribute und Methoden im „privaten“ Teil der Deklaration können nur in Methoden der Klasse verwendet werden.

– private = Zugriff nur in/durch Methoden der Klasse erlaubt

Eine erste Klasse

- Der Entwickler einer Klasse deklariert und definiert die Klasse und er implementiert die Methoden der Klasse.
- Es kann vorkommen, dass der Entwickler aufgrund von Problemen
  - die internen Datenstrukturen ändert muss oder dass
  - er einen effizienteren Algorithmus verwenden muss.
- Die öffentliche Schnittstelle (Signaturen und Rückgabewerte) sollte sich jedoch nicht ändern.
- Änderungen der obigen Art erfordern vom Nutzer nur, dass er das Programm neu übersetzt.
- Neue sinnvolle Methoden können natürlich im Verlauf der Weiterentwicklung einer Klasse hinzukommen.

Datenkapselung

- Um unzulässige Zugriffe und damit auch versehentliche Fehler zu vermeiden, sollte die tatsächliche Implementierung der Datenstrukturen einer Klasse nach außen nicht sichtbar sein.
- Man beachte: Die öffentliche Schnittstelle einer jeden Klasse sollte so klein wie möglich gehalten werden, d.h., alle internen Datenstrukturen sollten vor den Anwendern (Nutzern) einer Klasse versteckt werden. Diese Attribute sollten mit dem Schlüsselwort „private“ deklariert werden.
- Nur mittels „öffentlicher“ Elementfunktionen und über deren wohldefinierte Schnittstellen sollte ein Zugriff auf die Daten möglich sein. Diese sollten als „public“ deklariert werden.

Eine erste Klasse


```
// Klassendefinition in der Header-Datei  point2D.h
#ifndef  point2D_h
#define  point2D_h   point2D_h

class Point2D {
  public:
          int      x( );
          int      y( );
          void     setCoordinates(int, int);
          short    getColor(  );
          void     setColor(short);
  private:
          int      xCoordinate;
          int      yCoordinate;
          short    color;
  };
  #endif
```

Diese Präprozessor-Anweisungen verhindern, dass diese Header-Datei mehrfach eingebunden wird (mehr dazu in den Übungen).

Eine erste Klasse

– Die Klassendefinition mit den Deklarationen der Funktionen der Klasse findet man in einer Header-Datei, die den Namen der Klasse trägt (Beispiel: point2D.h).

– Die Implementierung der Funktionen der Klasse (Elementfunktionen) erfolgt gewöhnlich in einer Datei mit dem Namen „Klassenname.cpp“ (Beispiel: point2D.cpp).

```
// Datei point2D.cpp
// Der Header der Klasse muss eingebunden werden.
#include“point2D.h“
using namespace std;

int  Point2D::x()  { return xCoordinate; }
int  Point2D::y()  { return yCoordinate; }

//  Fortsetzung auf der nächsten Seite →
```

Eine erste Klasse


```
// Fortsetzung →

void Point2D::setCoordinates(int x, int y) {
        xCoordinate = x;
        yCoordinate = y;
}

short Point2D::getColor( )  { return color; }
void Point2D::setColor(short newColor) {  color = newColor; }

// Ende der Datei point2D.cpp
```

Durch den Klassennamen und den Bereichsoperator :: werden die obigen Funktionen als Methoden der Klasse „Point2D“ gekennzeichnet.

Eine erste Klasse

- Die Klassendeklaration beschreibt die Objekte einer Klasse. Sie erzeugt aber keine konkreten, tatsächlichen Objekte der Klasse.
- Die tatsächlichen Objekte heißen auch Instanzen einer Klasse.
- Wie kann man Objekte/Instanzen in einem Programm erzeugen?

```
Punkt2D  p1, p2;
```

- Tatsächliche Punkte p1 und p2 enthalten konkrete Daten.

Attribut	Werte für Punkt p1	Werte für Punkt p2
int xCoordinate	10	100
int yCoordinate	20	100
short color	1	3

Eine erste Klasse


```
// Ein Beispielprogramm:  Datei  beispiel_point2D.cpp
#include“point2D.h“
#include<iostream>
using namespace std;

int main( )  {
     Point2D  p1, p2;                  // zwei Punkte erzeugen
     p1.setCoordinates(10,20);         // Koordinaten von p1 setzen
     p1.setColor(3);                   // Farbe von p1 setzen
     p2.setColor(p1.getColor( ));      // Farbe von p2 setzen
     p1.setColor(1);                   // Farbe von p1 ändern

     p2.setCoordinates(100,100);       //  Koordinaten von p2 setzen

     cout  <<   p1.x( )  << p1.y( )  << p1.getColor()  << endl;
     cout  <<   p2.x( )  << p2.y( )  << p2.getColor()  << endl;
}
```

Eine erste Klasse

Wie erhalten wir ein lauffähiges Programm „beispiel_point2D“?

- point2D.cpp übersetzen → point2D.o
- beispiel_point2D.cpp übersetzen → beispiel_point2D.o
- point2D.o +beispiel_point2D.o linken

Die Datei „point2D.h“ wird während der Übersetzung der *.cpp-Dateien gelesen.

Bitte machen Sie sich mit dem Programm „make“ vertraut. Mit Make-Dateien kann man die Arbeit beim Übersetzen und Linken erheblich reduzieren.

Objekterzeugung und -benutzung

- Die Objekterzeugung ist wie die übliche Variablendefinition:

```
Point2D  p1, p2;
```

- Hierbei ist die Klasse der Datentyp.
- Bei jeder Objekterzeugung wird ein Konstruktor aufgerufen.
- Konstruktoren sind besondere Methoden einer Klasse, die Speicherplatz für Objekte reservieren und die diese eventuell auch initialisieren.
- Bei einfachen Klassen wie „Point2D“ kann der Programmierer die Deklaration und Implementierung eines Konstruktors dem Compiler überlassen.
- Explizite Deklarationen/Definitionen von Konstruktoren folgen.

Objekterzeugung und -benutzung

Attribut	Werte für Punkt p1	Werte für Punkt p2
int xCoordinate	10	
int yCoordinate	20	
short color	3	3

- Objektbenutzung:

```
p1.setCoordinates(10,20);        // Koordinaten von p1 setzen
p1.setColor(3);                  // Farbe von p1 setzen
p2.setColor(p1.getColor( ));     // Farbe von p2 setzen
```

```
void Point2D::setCoordinates(int x, int y) {
                    xCoordinate = x;
                    yCoordinate = y;
}

short Point2D::getColor( )  { return color; }
void Point2D::setColor(short newColor) {  color = newColor; }
```

Zugriffsrechte: „public“ und „private“


```
/* Da die Attribute „xCoordinate“, „yCoordinate“ und „color“ als
   „private“ deklariert wurden, sind die rot-markierten Zugriffe
   nicht erlaubt. */

int main( ) {
    Point2D  p1;

    p1.xCoordinate = 10;
    p1.yCoordinate = 20;
    p1.color = 3;

    cout  <<   p1.x( )  << p1.y( )  << p1.getColor()  << endl;
    }
```

Zugriffsrechte: „public“ und „private“


```
/*  Nur die Methoden der Klasse dürfen auf die privaten
    Attribute und Methoden zugreifen.                    */

#include“point2D.h“
using namespace std;

int  Point2D::x()  { return xCoordinate; }
int  Point2D::y()  { return yCoordinate; }

void Point2D::setCoordinates(int x, int y) {
        xCoordinate = x;
        yCoordinate = y;
}

short Point2D::getColor( )  { return color; }
void Point2D::setColor(short newColor) {  color = newColor; }
```

inline-Funktionen

– Funktionsaufrufe kosten Zeit, da
  - der Zustand des Aufrufers gesichert werden muss,
  - eventuell Parameter übergeben werden müssen und da
  - schließlich wieder der Aufrufer aktiviert werden muss.

– Inline-Funktionen vermeiden diesen Aufwand:

```
inline int quadrat(int x) { return x*x; }
```

– Beim Compilieren wird der Aufruf einer inline-Funktion durch den Funktionskörper ersetzt:

```
z = quadrat(100);   →   z = 100*100;
```

– Es erfolgt also kein wirklicher Funktionsaufruf.

Klassendefinition mit inline-Funktionen


```
// Klassendefinition mit inline-Funktionen
#ifndef  point2D_h
#define  point2D_h   point2D_h

class Point2D {
  public:
            int       x( ) { return xCoordinate; }
            int       y( ) { return yCoordinate; }
            void      setCoordinates(int x, int y) {
                              xCoordinate = x;
                              yCoordinate = y;
                      }
            short     getColor( )  { return color; }
            void      setColor(short newColor) {color = newColor;}
  private:
            int       xCoordinate;
            int       yCoordinate;
            short     color;
  };
  #endif
```

Klassendefinition mit inline-Funktionen

- Das Schlüsselwort „inline“ ist in Klassendefinitionen nicht erforderlich.
- Verwenden Sie inline-Funktionen nur bei sehr kleinen und einfachen Funktionen und Methoden.
- Durch die Textersetzungen kann es sonst zu schwer vorhersehbaren Fehlern kommen.

Konstruktoren

- Konstruktoren tragen den Namen ihrer Klasse:

```
Point2D::Point2D( ) {   /* Funktionskörper */  }
```

- Konstruktoren generieren neue Objekte, d.h.,
- sie reservieren Speicherplatz für ein neues Objekt und
- sie initialisieren gegebenenfalls die Daten des neuen Objekts.
- Falls in der Klassendefinition kein Konstruktor angegeben wird, so erzeugt der Compiler einen so genannten Default-Konstruktor.
- Der Default-Konstruktor kann auch vom Programmierer implementiert werden.
- Der Default- oder Standard-Konstruktor besitzt keine Parameter.

Konstruktoren

- Konstruktoren können wie andere Funktionen überladen werden:

```
Point2D::Point2D(int z ) {                  // zweiter Konstruktor
        xCoordinate = z;
        yCoordinate = 10;
}

Point2D::Point2D(int x, int y, short c) { // dritter Konstruktor
        xCoordinate = x;
        yCoordinate = y;
        color = c;
}
```

- Da alle Konstruktoren als Funktionsnamen den Namen der Klasse tragen, kann der Compiler den gewünschten Konstruktor nur anhand der Parameterliste identifizieren.

Konstruktoren

- Der Compiler wählt den passenden Konstruktor aus, in dem er Anzahl und Datentypen der Argumente der Parameterliste der Konstruktorendeklaration mit der Angabe im Aufruf vergleicht:

```
int x = 10, y = 20;
short incolor = 3;
Point2D  firstPoint;                    //  Default-Konstruktor
Point2D  secondPoint(x);                //  zweiter Konstruktor
Point2D  thirdPoint(50, y, incolor );   //  dritter  Konstruktor
```

- Man beachte: Wird ein allgemeiner Konstruktor deklariert und definiert, so muss auch der Default-Konstruktor deklariert und definiert werden.

Konstruktoren

- Was geschieht beim Aufruf eines Konstruktors?
  - Zunächst wird Speicherplatz für die Datenelemente reserviert:
    - xCoordinate
    - yCoordinate
    - color
  - Dann werden die Aktualparameter zugewiesen.
  - Schließlich wird der Programmcode innerhalb der geschweiften Klammern (Funktionskörper) ausgeführt.

Konstruktoren

- Initialisierungsliste: Eine effiziente Art der Initialisierung.

```
Point2D::Point2D(int x, int y, short c )
: xCoordinate(x), yCoordinate(y), color(c)      // Initialisierungsliste
{
  // in diesem Beispiel : leerer Block (Funktionskörper)
}
```

- Man beachte: Die Reihenfolge der Initialisierung richtet sich nach der Reihenfolge in der Deklaration der Klasse.

Konstruktoren

- Kopierkonstruktor (copy constructor):
  - Initialisierung eines Objektes mit den Daten eines anderen Objekts.
  - Argument = eine Referenz auf ein Objekt der entsprechenden Klasse:

```
Point2D::Point2D(const Point2D & pointToCopy)
        : xCoordinate(pointToCopy.xCoordinate),
         yCoordinate(pointToCopy.yCoordinate),
         color(pointToCopy.color)
        {
        // auch  in diesem Beispiel : leerer Block
        }
```

Konstruktoren

- Kopierkonstruktor (copy constructor)
  - wird nur dann verwendet, wenn ein neues Objekt generiert wird:

```
Point2D  firstPoint(19,39,1);
Point2D  secondPoint = firstPoint;
```

  - Man beachte: Im obigen Beispiel wird der Kopier-konstruktor verwendet (nicht der Zuweisungsoperator, siehe auch nächste Seite).

Konstruktoren


```
Point2D   p1, p2;                  // Default-Konstruktor
p1  =  Point2D(8,7);               // Allg. Konstruktor + Zuweisungsoperator
p2  =  p1;                         // Zuweisung
Point2D p3  = Point2D(1, 17);      // Ein temporäres Objekt wird
                                   // in das neu erzeugte Objekt kopiert.
```

Der Kopierkonstruktur wird nicht aufgerufen bei Zuweisungen oder Initialisierungen,

- bei denen ein temporär erzeugtes Objekt in das neu erzeugte Objekt kopiert wird.

Der Kopierkonstruktur wird aufgerufen, wenn

- ein Objekt an eine Funktion per Wert übergeben wird und
- bei der Rückgabe eines Ergebnisobjekts.

Konstruktoren

- Die Klassenbeschreibung von Point2D mit allen angegebenen Konstruktoren hat die folgende Form:

```
class Point2D  {
        public:
                Point2D( );                              // Default-Konstruktor
                Point2D(int x);                          // Allg. Konstruktor 1
                Point2D(int x, int y, short c);          // Allg. Konstruktor 2
                Point2D(const Point2D  &pointToCopy);    // Kopierk.
                int     x();
                int     y();
                void    setCoordinates(int x, int y);
                short   getColor( );
                void    setColor(short);
        private:
                int     xCoordinate;
                int     yCoordinate;
                short   color;
    };
```

Zuweisungsoperator =

- Falls kein spezieller Zuweisungsoperator in einer Klasse definiert ist, wird automatisch vom Compiler ein Zuweisungsoperator bereitgestellt.
- Hierbei wird für jedes Element, das selbst Objekt oder aber von einem Grunddatentyp ist, der zugehörige Zuweisungsoperator aufgerufen.
- Beispiel für einen Zuweisungsoperator der Klasse Point2D:

```
Point2D& Point2D::operator=(const Point2D & secondPoint) {
   // Zuweisung identischer Objekte vermeiden
   if (this != &secondPoint) {     // this ist ein Zeiger auf das aktuelle Objekt
     this->xCoordinate = secondPoint.xCoordinate;
     this->yCoordinate = secondPoint.yCoordinate;
     this->color = secondPoint.color;
   }
   return *this;                    //   *this ist das Objekt selbst
}
```

Destruktoren

- Destruktoren sind Elementfunktionen, die den Speicherplatz von nicht mehr benötigten Elementen wieder freigeben.
- Falls kein Destruktor für eine Klasse deklariert und definiert wird, erzeugt der Compiler automatisch einen Destruktor.
- Automatisch erzeugte Destruktoren von Klassen mit dynamischen Strukturen geben in der Regel nicht den gesamten Speicherplatz wieder frei (memory leaks).

Destruktoren

- Destruktoren besitzen keine Argumente und keinen Rückgabetyp.
- In der Deklaration wird ein Tilde ~ vorangestellt:

```
class Point2D  {
        public:
                Point2D( );                             // Default-Konstruktor
                Point2D(int x);                         // Allg. Konstruktor 1
                Point2D(int x, int y, short c);         // Allg. Konstruktor 2
                Point2D(const Point2D  &pointToCopy);       // Kopierk.
                ~Point2D( );                            // Destruktor
                int     x();
                int     y();
                void    setCoordinates(int x, int y);
                short   getColor( );
                void    setColor(short);
        private:
                int     xCoordinate;
                int     yCoordinate;
                short   color;
    };
```

Destruktoren und Gültigkeitsbereiche

- Die Gültigkeit oder Lebensdauer eines Objekts bzw. einer Variablen endet, an dem mit der schließenden geschweiften Klammer markierten Ende des Blocks, in dem das Objekt oder die Variable definiert wurde.
- Dann ruft der Compiler den Destruktor für das Objekt auf.

```
{
   // äußerer Block ....
   Point2D p1;
   // äußerer Block ....
    {
      //  innerer Block mit Anweisungen und Deklarationen
      Point2D p2;
     //  innerer Block
    }                          <-- Compiler ruft den Destruktor für p2 auf
   // äußerer Block .....
}                              <-- Compiler ruft den Destruktor für p1 auf
```

Destruktoren und Gültigkeitsbereiche

- Aufruf des Destruktors durch den Programmierer:

```
delete p1;
delete p2;
```

- Bei Feldern a[ ] und Vektoren v steht nach dem Schlüsselwort „delete“ noch der Klammeroperator:

```
delete [ ] a;
delete [ ] v;
```

const-Objekte und Methoden

- Objekte können als konstant deklariert werden:

```
const  Point2D  pconst(10,10,1);
```

- Methoden dürfen von konstanten Objekten nicht aufgerufen werden. Ausgenommen sind
  - Konstruktoren
  - Destruktoren
  - Methoden, die als „const“ deklariert und definiert wurden:

```
Rückgabetyp  Funktionsname(Parameterliste)   const;
```

```
Rückgabetyp  Funktionsname(Aktual-Parameterliste) const
        {    /* Funktionskörper */   }
```

const-Objekte und Methoden

- Solche Methoden besitzen das Privileg, für konstante und nicht-konstante Objekte aufgerufen werden zu können.

```
class Point2D  {
        public:
                Point2D( );                                  // Default-Konstruktor
                Point2D(int x);                              // Allg. Konstruktor 1
                Point2D(int x, int y, short c);              // Allg. Konstruktor 2
                Point2D(const Point2D  &pointToCopy);        // Kopierkonstruktor
                ~Point2D( );                                 // Destruktor
                int     x() const;
                int     y() const;
                void    setCoordinates(int x, int y);
                short   getColor( ) const;
                void    setColor(short);
        private:
                int     xCoordinate;
                int     yCoordinate;
                short   color;
    };
```

Die const-Methoden dürfen den Zustand eines Objekts nicht verändern.

Klassentemplates

- Template Klassen = parametrisierte Datentypen
- Beispiel: Einen parametrisierten Datentyp für Klasse Point2D:

```
// Point2D.t
#ifndef Point2D_t
#define Point2D_t

template<typename T>
class Point2D  {
        public:
            Point2D( );                                  // Default-Konstruktor
            Point2D(T  x);                               // Allg. Konstruktor 1
            Point2D(T  x, T  y, short c);                // Allg. Konstruktor 2
            Point2D(const Point2D  &pointToCopy);   // Kopierkonstruktor
            ~Point2D( );                                 // Destruktor
            // Fortsetzung auf der nächsten Seite →
```

Klassentemplates


```
            // Fortsetzung von vorhergehender Folie
            T       x() const;
            T       y() const;
            void    setCoordinates(T x, T y);
            short   getColor( ) const;
            void    setColor(short);
        private:
            T       xCoordinate;
            T       yCoordinate;
            short   color;
    };

#endif
```

- <typename T> anstelle von <class T>

Klassentemplates


```
// Implementierung von einigen Methoden
template<typename T>
T   Point2D<T>::x( ) const { return xCoordinate; }

template<typename T>
T   Point2D<T>::y( ) const { return yCoordinate; }

template<typename T>
void Point2D<T>::setCoordinates( T x , T y) {
      xCoordinate = x;
      yCoordinate = y;
}

// usw.
```

Klassentemplates

Wie generiert man Objekte einer parametrisierten Klasse?

```
// Beispiel für Objektgenerierung
#include“point2D.t“
using namespace std;

int main( )  {
   Point2D<int>  pint;
   Point2D<float> pfloat(1.5, 2.5, 3);
   pint.setColor(pfloat.getColor());
   pint.setCoordinates((int)pfloat.x(), (int)pfloat.y());
}
```

Vererbung

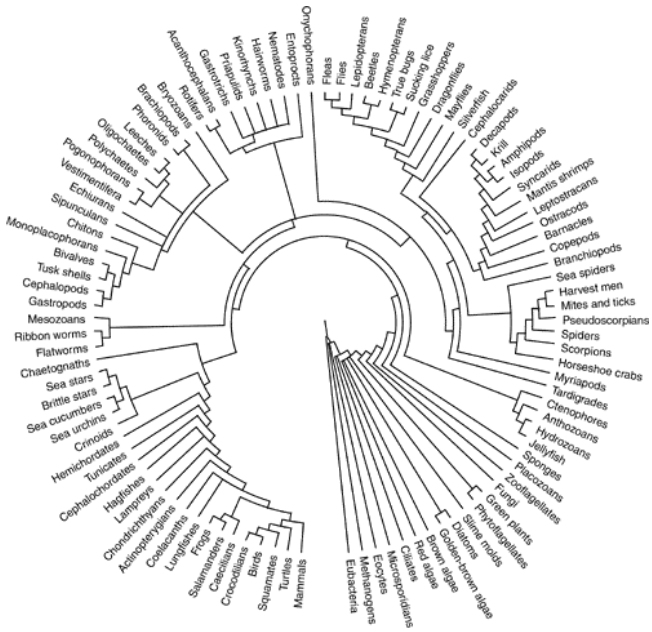
Wiederverwendbarkeit von Programmcode ist das zentrale Ziel des OOP!

Ein wichtigstes Konzept zur Unterstützung der Wiederverwendbarkeit von Programmcode ist die Vererbung!

Vererbung

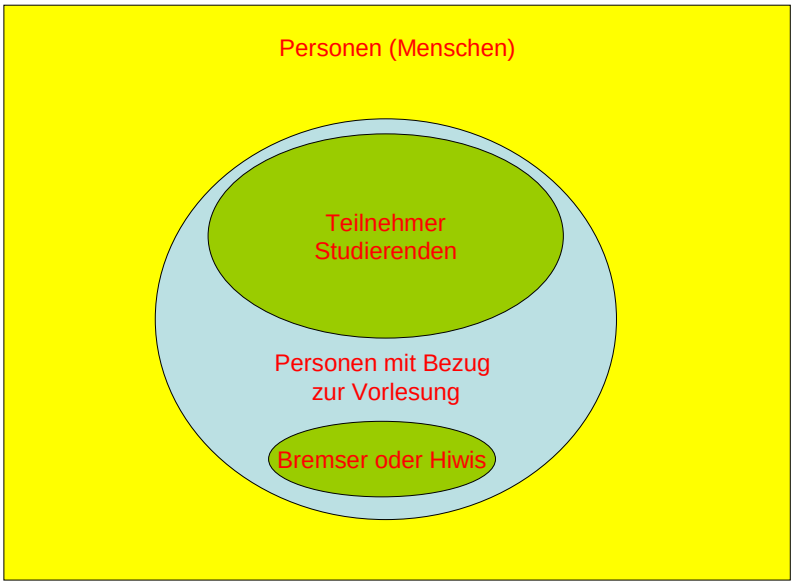
- Was versteht man unter Vererbung?
  - Klassen beschreiben Objekte mit ähnlichen Eigenschaften und ähnlichen Verhaltensweisen (Methoden).
  - Die Objekte einer Klasse können sich dennoch in bestimmten Eigenschaften und Verhaltenweisen unterscheiden.
  - Es kann zum Beispiel Unterklassen geben, die besondere Eigenschaften und Verhaltensweisen besitzen.

Vererbung



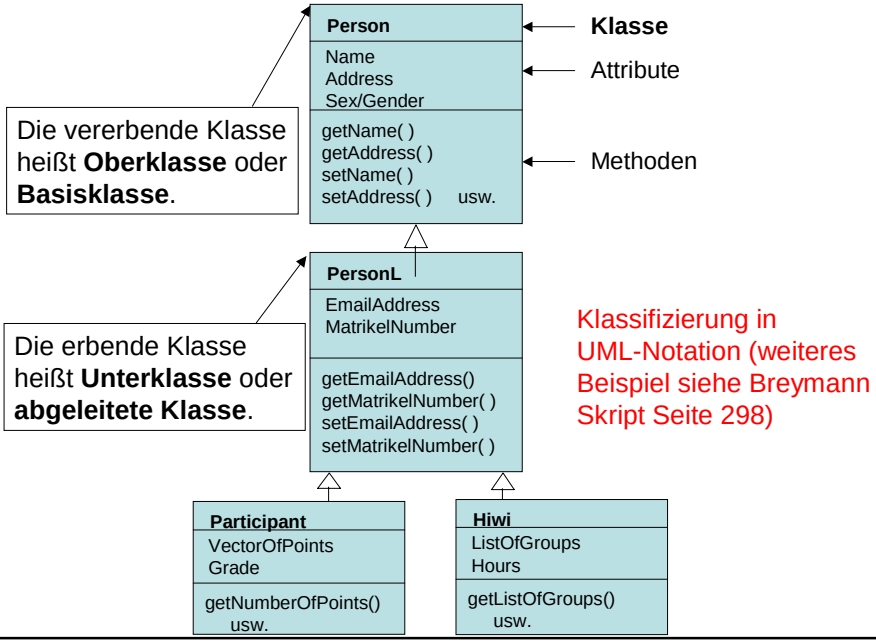
Vererbung

47

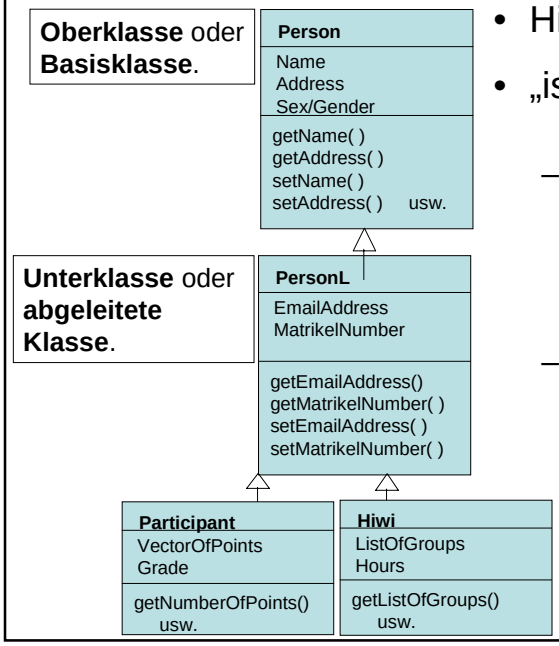


Vererbung

48

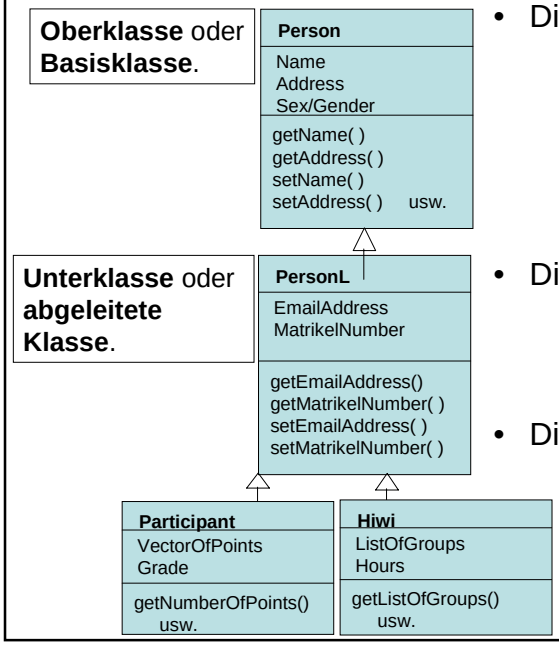


Vererbung und Klassifizierung



- Hierarchie
- „ist-ein-Beziehung“, d.h.,
  - Ein Objekt der abgeleiteten Klasse ist gleichzeitig auch ein Objekt der Basisklasse.
  - Die Menge der Objekte in der abgeleiteten Klasse sind daher eine Teilmenge der Objekte der Basisklasse.

Vererbung und Klassifizierung



- Die abgeleitete Klasse erbt
  - alle Eigenschaften (Attribute) und
  - Verhaltensweisen (Methoden) der Basisklasse.
- Die Unterklasse ist eine **Spezialisierung** der Oberklasse.
- Die Oberklasse ist die **Abstraktion** oder **Generalisierung** der Unterklassen.

Vererbung und Klassifizierung

- Wenn eine Oberklasse bekannt ist, so brauchen in der abgeleiteten Klasse nur die Abweichungen beschrieben zu werden.
- Ist eine Oberklasse implementiert, so müssen wir für die „neue“ abgeleitete Klasse nur die speziellen Attribute und Methoden der abgeleiteten Klasse implementieren.

→ Alles andere kann wieder verwendet werden.

Vererbung und Klassifizierung

- Nehmen wir an, dass wir die Basisklasse „Person“ bereits definiert und implementiert hätten:

```
class Person {
   public:
      string  getName( );                    // den Namen ausgeben
      void    setName(string pname);         // den Namen setzen
      string  getAddress( );                 // die Adresse ausgeben
      void    setAddress(string paddress);   // die Adresse
   private:
      string  name;
      string  address;
};
```

- Die Implementierung der Methoden der Klasse „Person“ geben wir hier nicht an.

Vererbung und Klassifizierung

- So können wir die Klasse „PersonL(ecture)“ wie folgt von der Klasse „Person“ ableiten:

```
class PersonL : public Person {
   public:
      string             getEmailAddress();
      void               setEmailAddress(string email);
      unsigned long getMatrikelNumber( );
      void               setMatrikelNumber(unsigned long matrikelNo);
   private:
      string              emailAddress;
      unsigned long  matrikelNumber;
};
```

- Syntax:

  class Unterklassenname : public Oberklassenname

Vererbung und Klassifizierung

- Von der Klasse PersonL können wir anschließend die Klasse „Participant“ (und die Klasse Hiwi) ableiten:

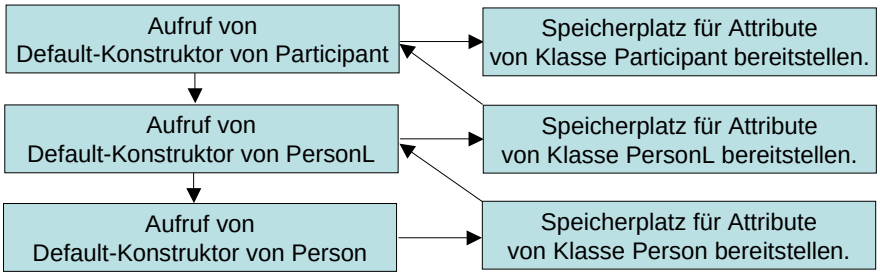
```
class Participant: public PersonL {
   public:
      int                getNumberofPoints();    // Punktezahl berechnen
      void               setGrade(float  note);  // Note eingeben
      float              getGrade( );            // Note zurückgeben
   private:
      vector<int>      vectorOfPoints;
      float              grade;
};
```

- Diese abgeleitete Klasse erbt nun nicht nur von der Klasse PersonL, sondern natürlich auch von der Basisklasse von PersonL (also von Person).

Vererbung und Klassifizierung


```
Participant   paul;
```

- Die Konstruktoren werden in der Reihenfolge der Vererbungshierarchie implizit aufgerufen:



- Konstruktoren der Basisklassen können auch explizit in selbst definierten Konstruktoren der abgeleiteten Klasse aufgerufen werden.

Vererbung und Klassifizierung

- Definiert man ein Objekt der Klasse „Participant“, so kann man alle öffentlichen Elementfunktionen der Klasse
  - Participant und der Basisklassen
  - Person, PersonL auf dieses Objekt anwenden.

```
Participant  paul;
paul.setName(“Paul Hacker“);                    // Person
paul.setEmailAddress(“paul@rz.uni-sb.de“);   // PersonL

if (paul.getGrade( ) <= 4.0)  cout << “Paul hat bestanden!“ << endl;
```

Vererbung und Klassifizierung

- Konstruktoren der Basisklassen können auch explizit in selbst definierten Konstruktoren der abgeleiteten Klasse aufgerufen werden:
  - Entweder im Funktionskörper:

```
Participant::Participant(string pname, string email, unsigned long matNo)
{
   PersonL(email, matNo);   // Hierfür müsste ein entsprechender
                            // allgemeiner Konstruktor mit diesen
                            // Parametern definiert werden.
   this->setName(pname);
}
```

  - Oder in der Initialisierungsliste:

```
Participant::Participant(string pname, string email, unsigned long matNo)
: PersonL(email, matNo)  {
 this->setName(pname);
}
```

Zugriffsschutz

- Unter Zugriffschutz ist die Abstufung von Zugriffsrechten auf Daten und Elementfunktionen zu verstehen.
- Bisher haben wir zwei Abstufungen kennen gelernt:
  - public:

    Elemente und Methoden unterliegen keiner Zugriffsbeschränkungen.
  - private:

    Elemente und Methoden sind ausschließlich innerhalb der Klasse zugreifbar, sowie für friend Klassen und –Funktionen.

Zugriffsschutz

- Um abgeleiteten Klassen weitergehende Rechte einräumen zu können, ohne den privaten Status mancher Elemente aufzugeben, gibt es einen weiteren Zugriffsspezifizierer:

  – protected:

    Elemente und Methoden sind in der eigenen und in allen mit „public“ abgeleiteten Klassen zugreifbar, nicht aber in anderen Klassen oder außerhalb der Klasse.

Zugriffsschutz

Vererbung von Zugriffsrechten:

– Gegeben sei eine Oberklasse, von der eine weitere Klasse abgeleitet wird.

– Für die Vererbung der Zugriffsrechte bei „public“-Vererbung gelten folgende Regeln:

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

Zugriffsschutz

Regeln für die Vererbung (public, private, protected):

- Private Elemente sind in einer abgeleiteten Klasse nicht zugreifbar.
- In allen anderen Fällen gilt das jeweils restriktivere Zugriffsrecht, bezogen auf die Zugriffsrechte für ein Element und die Zugriffskennung der Vererbung einer Klasse (siehe Beispiel im Breymann-Skript Seite 310-312).

Polymorphismus

- **Polymorphismus = Vielgestaltigkeit**
- Fähigkeit, erst zur Laufzeit eines Programms die zu dem jeweiligen Objekt passende Realisierung einer Operation zu ermitteln:
  - Ein Funktionsaufruf muss irgendwann an eine Folge von auszuführenden Anweisungen gebunden werden.
  - während der Ausführung des Programms →**dynamisches Binden**.
  - vor der Ausführung des Programms →**statisches Binden**.
- Eine zur Laufzeit ausgewählte Methode heißt **virtuelle Funktion**.

Polymorphismus: Beispiel

- aus dem Bereich Computergraphik:
- Wir betrachten 3 Klassen, eine Basisklasse
  - „GraphObj“ und zwei von „GraphObj“ abgeleitete Klassen
  - „Square“ und „Circle“.
- Die drei Klassen haben eine Funktion „drawObj“
  - mit dem gleichem Rückgabetyp „void“ und
  - den gleichen Eingabeparametern.
- „drawObj“ soll die graphischen Objekte auf dem Bildschirm zeichnen.

Polymorphismus: Beispiel


```
class  GraphObj {
        public:
            virtual void drawObj( ) const;
             // und weitere Methoden
        private:
             // und Elemente
};
```

```
class  Square : public GraphObj {
        public:
            virtual void drawObj( ) const;
            // und weitere Methoden
        private:
            // und Elemente
};
```

```
class  Circle: public GraphObj {
        public:
            virtual void drawObj( ) const;
            // und weitere Methoden
        private:
            // und Elemente
};
```

Durch Vorstellen des Schlüsselworts „virtual“ teilen wir dem Compiler mit, dass es sich um virtuelle Funktionen handelt.

Polymorphismus: Beispiel

- Ein Graphikprogramm produziert eine Reihe von Quadraten und Kreisen in beliebiger Reihenfolge:

```
vector<GraphObj*>  vectorOfSquaresAndCircles;
```

- Wir zeichnen nun alle Kreise und Quadrate in einer for-Schleife:

```
for (size_t  i = 0; i < vectorOfSquaresAndCircles.size(); i++)
        vectorOfSquaresAndCircles[i]->drawObj();
```

- Erst zur Laufzeit (der for-Schleife) wird die richtige Zeichenfunktion für jedes im Vektor gespeicherte graphische Objekt dynamisch gebunden werden.

Polymorphismus

- Dynamisches Binden mittels virtueller Funktionen?
  - „virtual“ bewirkt, dass
    - Objekten Information über den Objekttyp mitgegeben wird,
    - in Form eines Zeigers „vptr“ auf eine Tabelle „vtbl“ (virtual table) mit Zeigern auf die zugehörigen virtuellen Funktionen.
  - Diese Tabelle gehört zu der Klasse des Objekts.
  - Hat die Klasse des Objekts keine passende Signatur, so wird eine entsprechende Funktion der Oberklasse gesucht.

Polymorphismus

- Virtuelle Funktionen dienen zum Überladen bei gleicher Signatur und bei gleichem Rückgabetyp.
- Der Einsatz virtueller Funktionen bewirkt, dass Objekten die Typinformation mitgegeben wird.
- Die Objekte werden durch den Zeiger „vptr“ etwas größer.
- Der Umweg über den Zeiger kostet natürlich auch Rechenzeit.
- Dynamisches Binden sollte nur dann verwendet werden, wenn die Klassen der Objekte erst zur Laufzeit bekannt werden (und nicht schon beim Übersetzen feststehen).

Abstrakte Klassen

- Basisklassen sollten in der Regel sehr allgemein sein.
- Oft ist es nicht notwendig, dass Objekte dieser generellen Basisklassen angelegt werden, da alle für die Aufgaben-stellung interessanten Objekte nur von den abgeleiteten Klassen erzeugt werden.
- Diese abstrakten Klassen dienen ausschließlich als Ober- oder Basisklassen.
- Beispiel: GraphObj → Circle, Square

Abstrakte Klassen

Durch Ergänzung „ = 0“ wird die Funktion zur rein virtuellen Funktion.

```
class  GraphObj {
        public:
            virtual void drawObj( ) const = 0;
            // und weitere Methoden
        private:
            // und Elemente
};
```

```
class  Square : public GraphObj {
        public:
            virtual void drawObj( ) const;
            // und weitere Methoden
        private:
            // und Elemente
};
```

```
class  Circle: public GraphObj {
        public:
            virtual void drawObj( ) const;
            // und weitere Methoden
        private:
            // und Elemente
};
```

- Das syntaktische Mittel, um eine Klasse zur abstrakten Klasse zu machen, sind rein virtuelle Funktionen.
- Abstrakte Klassen haben mindestens eine rein virtuelle Funktion.

Abstrakte Klassen

- Klassen, von denen Objekte erzeugt werden können, nennt man **konkrete Klassen**.
- Klassen, von denen keine Objekte erzeugt werden können, nennt man **abstrakte Klassen**.
- Rein virtuelle Funktionen besitzen keine Implementierung in der abstrakten Klasse.
- Wenn eine konkrete Klasse von einer abstrakten Klasse abgleitet ist, so muss sie konkrete Implementierungen für die rein virtuellen Funktionsprototypen zur Verfügung stellen.
- Fehlt die konkrete Implementierung einer rein virtuellen Funktion in einer abgeleiteten Klasse, so ist die Klasse auch eine abstrakte Klasse.

Mehrfachvererbung

– Eine Klasse kann von mehreren anderen Klassen abgeleitet werden und „erben“:

```
class Multierbe: public Basisklasse1, public Basisklasse2  {
      //  das Übliche
}
```

– Die Basisklassen werden durch Kommata getrennt.
– Für Beispiele siehe Breymann-Skript Seite 341++.

Hat-Beziehung oder Aggregation

– Ein Objekt einer Klasse kann Objekte von anderen Klassen als Teile enthalten.
– Beispiel: Die Linie **hat** einen Anfangs- und einen Endpunkt:

```
#include “graphObj.h“
#include  “point2D.h“

class Line: public GraphObj  {
      public:
            Point2D&  getStartPoint( );
            Point2D&  getEndPoint( );
            virtual  drawObj( ) const;
            // plus weitere nützliche Methoden
      private:
            Point2D  startPoint;
            Point2D  endPoint;
  }
```

Überladen von Operatoren

– Den vordefinierten Operatorsymbolen in C++ kann man für selbst implementierte Klassen neue Bedeutungen zuordnen.

– Dies ermöglicht es dem Nutzer gewisse Operationen sehr kompakt zu schreiben.

– Wir sprechen wie bei Funktionen auch bei den Operatoren vom **Überladen** (von Operatoren).

– Überladen werden können die üblichen Operatoren in C++, wie zum Beispiel

= == > < <= += + * / << >> usw.

mit Ausnahme von . .* :: ?:

Überladen von Operatoren

– Wir verwenden das Symbol ⊗ als Platzhalter für die verschiedenen Operatoren in C++.

– Bei einer globalen Funktion ist die Syntax einer Operatordefinition

```
Rückgabetyp  operator⊗(Argumentliste) {  Funktionsblock }
```

– Bei einer Elementfunktion ist die Syntax einer Operatordefinition

```
Rückgabetyp Klassenname::operator⊗(Argumentliste)
                                    { Funktionsblock }
```

– Für die Syntax von Operatoren und die Funktionsaufrufe, die der Compiler mittels der Operatoren tätigt, siehe Seite 369 im Breymann-Skript.

Überladen von Operatoren

Element-Funktion	Syntax	Ersetzung durch
nein	x⊗y	operator ⊗(x,y)
	⊗x	operator ⊗(x)
	x⊗	operator ⊗(x,0)
ja	x⊗y	x.operator ⊗(y)
	⊗x	x.operator ⊗()
	x⊗	x.operator ⊗(0)
	x=y	x.operator=(y)
	x(y)	x.operator()(y)
	x[y]	x.operator[](y)
	x->	(x.operator->())->
	(T)x	x.operator T();

Klassentemplates

- Als Beispiel fügen wir zu unserem Klassentemplate Point2D<T> Operatoren hinzu, so dass wir mit den zweidimensionalen Punkten wie mit komplexen Zahlen rechnen können.

```
// Point2D.t
#ifndef Point2D_t
#define Point2D_t

template<typename T>
class Point2D  {
        public:
                Point2D( );                                     // Default-Konstruktor
                Point2D(T  x);                                  // Allg. Konstruktor 1
                Point2D(T  x, T  y, short c);                   // Allg. Konstruktor 2
                Point2D(const Point2D  &pointToCopy);       // Kopierk.
                ~Point2D( );                                    // Destruktor
                // Fortsetzung auf der nächsten Seite →
```

Klassentemplates


```
            // Fortsetzung von vorhergehender Folie
            T               x() const;
            T               y() const;
            void            setCoordinates(T x, T y);
            short           getColor( ) const;
            void            setColor(short);
            Point2D<T>      operator+(const Point2D<T>& p);
            Point2D<T>      operator*(const  Point2D<T>& p);
            bool            operator==(const Point2D<T>& p);
        private:
            T               xCoordinate;
            T               yCoordinate;
            short           color;
    };
// globale Funktion zur Ausgabe
template<typename T>
std::ostream&               operator<<(std::ostream&, const Point2D<T>&);
#endif
```

Überladen von Operatoren

– Implementierungen der Operatoren:

```
template<typename T>
Point2D<T>  Point2D<T>::operator+(const Point2D<T>& p)
 {
   Point2D<T> sum;
   sum.xCoordinate = this->xCoordinate + p.xCoordinate;
   sum.yCoordinate = this->yCoordinate + p.yCoordinate;
   return sum;
}
```

Überladen von Operatoren


```
// Multiplikation
template<typename T>
Point2D<T>  Point2D<T>::operator*(const Point2D<T>& p)
{
   Point2D<T> product;
   product.xCoordinate = this->xCoordinate * p.xCoordinate –
                         this->yCoordinate * p.yCoordinate;
   product.yCoordinate = this->xCoordinate * p.yCoordinate +
                         this->yCoordinate * p.xCoordinate;
   return product;
}
```

Überladen von Operatoren


```
// Vergleichsoperator ==
bool  Point2D<T>::operator==(const Point2D<T>& p) {
    return ((this->xCoordinate == p.xCoordinate) &&
            (this->yCoordinate == p.yCoordinate));
}

// Ausgabe: ist keine Methode von Point2D<T>,
// sondern eine globale Funktion
std::ostream& operator<<(std::ostream& s,
                         const Point2D<T>& p)
{
  s  << "(" <<  p.X() << "," << p.Y() << "," << p.color << ")";
  return s;
}
```

Überladen von Operatoren

– Wenn wir die Koordinaten eine Punktes p (Point2D) mit dem Befehl

```
cout << p;
```

– ausgeben wollen, so können wir den Operator << nur als eine globale Funktion deklarieren und definieren, denn ansonsten müsste die Operatorfunktion eine Elementfunktion der Klasse ostream sein:

```
cout.operator<<(p);   // Fehler, da eine solche Elementfunktion mit
                      // Parameter Point2D<T> in der Klasse
                      //  ostream nicht existiert.
```

– Folglich haben wir für die Syntax cout << p (cout ⊗ p = x ⊗ y) nur die Möglichkeit eine globale Operatorfunktion zu definieren:

```
std::ostream& operator<<(std::ostream& , const Point2D<T>& ) ;
```