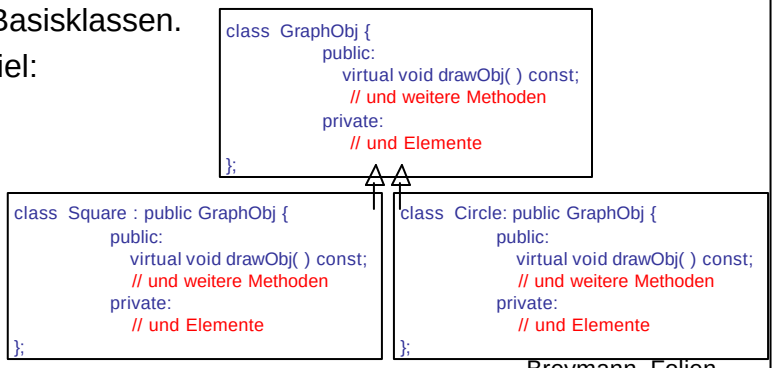
Abstrakte Klassen

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



Breymann_Folien

Abstrakte Klassen

- Bei der 3D-Darstellung von gewissen Szenen mittels der Computergraphik würde man nur konkrete graphische Objekte, die man zeichnen kann, also Quadrate, Kreise (Dreiecke: eine weitere Unterklasse von GraphObj) generieren und zeichnen.
- Es würde kein „konkretes“ Objekt der Klasse GraphObj generiert werden.
- Das syntaktische Mittel um eine Klasse zur abstrakten Klasse zu machen, sind rein virtuelle Funktionen.
- Abstrakte Klassen haben mindestens eine rein virtuelle Funktion:

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

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

Breymann_Folien

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 Implementierungen 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.
- Man beachte: 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++.

Teil-Ganzes- oder Hat-Beziehung

5

- oder Aggregation im Englischen:
- Ein Objekt einer Klasse kann Objekte von anderen Klassen als Teile enthalten.
- Als Beispiel betrachten wir hier eine (zweidimensionale) Linie mit einem Anfangs- und einem Endpunkt:

```
#include "graphObj.h"
#include  "point2D.h"

class Line: public GraphObj  {
      public:
            Point2D&  getStartPoint( );
            Point2D&  getEndPoint( );
            virtual  drawObj( ) const;
            // plus other useful methods
      private:
            Point2D  startPoint;
            Point2D  endPoint;
  }
```

- Man spricht von einer „Hat"-Beziehung, da
  - die Linie einen Anfangs- und einen Endpunkt **hat**.
- Eine Linie ist ein graphisches Objekt . Deshalb wird die Klasse Line von der Klasse GraphObj abgeleitet.

Breymann_Folien

Überladen von Operatoren

6

- Den vordefinierten Operatorsymbolen in C++ kann man für 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 . .* :: ?:

- Wir verwenden das Symbol ? als Platzhalter für die verschiedenen Operatoren in C++.
- Bei einer globalen Funktion ist die Syntax eines Operatoraufrufs

  Rückgabetyp operator? (Argumentliste) { Funktionsblock }

- Bei einer Elementfunktion ist die Syntax eines Operatoraufrufs

  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.

Breymann_Folien

Überladen von Operatoren

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

```
template<typename T>
class Point2D {
   public:
          T X( ) const;
          T Y( ) const;
          void changeCoordinates ( T x , T y);
          Point2D<T>  operator+(const Point2D<T>& p);    // neuer Operator +
          Point2D<T>  operator*(const Point2D<T>& p);    // neuer Operator *
          bool          operator==(const Point2D<T>& p);  // neuer Operator ==
   private:
          T  xCoordinate;
          T  yCoordinate;
};

// 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
ostream& operator<<(ostream& s, const Point2D<T>& p) {
  s  <<  "("  <<  p.X() << ","  << p.Y() << ")";
  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>& ) ;
```

Breymann_Folien