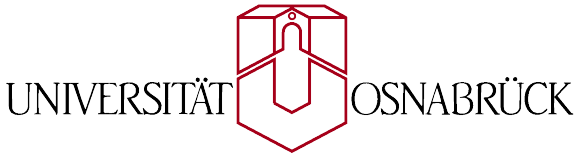
Einführung in die Programmiersprache C++

Dr. Thomas Wiemann
Institut für Informatik
AG Wissensbasierte Systeme

WS 2014 / 2015



Letzte Vorlesung

- Wo implementiere ich die neuen Operatoren?
- **Variante 1:** Als Teil der Klasse:

```
class Point{
   ...
   const Point operator+(const Point &p);
   ...
}
```

- Der erste Operand wird durch die Klasse repräsentiert (`this`)
- **Variante 2:** Außerhalb der Klasse:

```
Point operator+(const Point &p1, const Point &p2);
```

- Wird dann benötigt, wenn der erste Operand einen anderen Typ hat:

```
Point p;
p * 5 // Ok, geht in Klasse
5 * p // Geht nur außerhalb
```

Memberinitialisierung (1)

- Bisher haben wir Konstruktoren wie folgt geschrieben:

```
Point::Point(double x, double y)
{
  x_coord = x;     // Store x and y values.
  y_coord = y;
}
```

- Frage: Warum funktioniert das?
- Wie werden `x_coord` und `y_coord` erzeugt?
- Antwort: Alle Klassenvariablen werden erzeugt, bevor der Code des Konstruktors ausgeführt wird. Beispiel:

```
class GraphicsEngine {
  Matrix viewportTransform;
  Matrix modelViewTransform;
  ...
public:
  GraphicsEngine();
  ...
};
```

Die Konstruktoren für Matrix werden vor dem Konstruktor der Klasse `GraphicsEngine` aufgerufen

Memberinitialisierung (2)

- Nach unserer Vorgehensweise würde der Konstruktor für `GraphicsEngine` so aussehen:

```
GraphicsEngine::GraphicsEngine()
{
  viewportTransform = Matrix(4, 4);
  modelViewTransform = Matrix(4, 4);
  ... // Rest of graphics-engine initialization
}
```

- Aber die Matrizen wurden schon erzeugt
- Dieser Code ist ineffizient:

1. 2 mal Default-Matrix-Konstruktor
2. 2 mal Zwei-Argumente-Matrix-Konstruktor
3. 2 mal Zuweisungs-Operator (Copy / Cleanup)

- Wir wollen eigentlich nur Nummer 2 machen
- So genannten Member Initializer Lists lösen das Problem

Memberinitialisierung (3)

- In den Initialisierungslisten kann man festlegen, welche Konstruktoren für Member aufgerufen werden

```
GraphicsEngine::GraphicsEngine() :
viewportTransform(4, 4), modelViewTransform(4, 4)
{
  ... // Rest of graphics-engine initialization
}
```

- Nach der Konstruktor-Signatur und vor dem Code des Konstruktors
- Beachte:
  - Doppelpunkt vor der Initializer-Liste
  - Liste durch Kommata getrennt

Memberinitialisierung (4)

- Am gebräuchlichsten sind MILs für Member-Klassen
- Dann großer Performanzgewinn
- Einige Member benötigen Initializer, z.B.
  - Member-Objekte ohne Default-Konstruktor
  - const-Member
  - Referenzen

Gliederung

C++ Dynamische Speicherbereitstellung (1)

- In C gab es `malloc()` / `calloc()` um Speicher bereitzustellen und `free()` um ihn wieder frei zu geben
- Die C++-Befehle dazu sind `new` und `delete`
- `new` und `delete` sind keine Funktionen
- Rückgabewert von `new` sind Pointer mit einem Typ
- Beispiel:

p ist ein Pointer zu der Instanz von Point

```
Point *p = new Point(3.5, 2.1);
p->setX(3.8);              // Use the point...
delete p;                  // Free the point
```

- Lokale Variablen: Destruktor wird automatisch aufgerufen

```
void doStuff() {
  Point a(-4.75, 2.3); // Make a Point...
}
```

Destruktor wird hier aufgerufen

C++ Dynamische Speicherbereitstellung (2)

- Allokierte Objekte müssen selbst aufgeräumt werden
- Beispiel:

```
void leakMemory() {
   Point *p = new Point(-4.75, 2.3);
   ...
}
```

Der Pointer ist weg, das Objekt ist noch vorhanden!

- Das Objekt wird erst durch einen expliziten Aufruf von `delete ptrToObj` gelöscht
- Arrays von Objekten (statisch):

```
Point tenPoints[10];          // Index 0..9
tenPoints[3].setX(21.78);     // Fourth element
```

- Default Destruktor wird für jedes Objekt aufgerufen

C++ Dynamische Speicherbereitstellung (3)

- Arrays von Objekten (dynamisch mit `new`-Operator):

```
Point *somePoints = new Point[8];  // Index 0..7
somePoints[5].setY(-14.4);         // 6th element
```

- Dynamisch erzeugte Arrays müssen mit dem `delete[]`-Operator gelöscht werden:

```
delete[] somePoints; // Clean up my Points
```

- Compiler verhindert ein einfaches `delete` nicht!
- Man kann `new/delete[]` auch auf einfache Datentypen anwenden:

```
int numValues = 35;
int *valArray = new int[numValues];
```

- Felder sind nicht initialisiert

C++ Dynamische Speicherbereitstellung (4)

- Wenn eine Klasse dynamisch allokierten Speicher verwendet:
  - Falls möglich: Speicherbereitstellung im Konstruktor
  - Speicherfreigabe im Destuktor
- Beispiel:

```
// Initialize a vector of n floats.
FloatVector::FloatVector(int n)
{
  numElems = n;
  elems = new float[numElems];
}

// Clean up the float-vector.
FloatVector::~FloatVector()
{
  delete[] elems; // Release the memory for the array.
}
```

C++ Dynamische Speicherbereitstellung (5)

- Nun räumt `FloatVector` selbst auf:

```
float calculate() {
  FloatVector fvect(100); // Use our vector
  ...
}
```

- `fvect` ist eine lokale Variable
- Der Destruktor gibt dynamisch angeforderten Speicher wieder frei
- Hier wollen wir bestimmt nicht den default-Destruktor!

C++ Copy-Konstruktor (1)

- Beispiel Punkt-Klasse (`Point.hh`)

```
// A 2D point class!
class Point {
  double x_coord, y_coord;   // Data-members
public:
  Point();                   // Constructors
  Point(double x, double y);
  ~Point();                  // Destructor
  double getX();             // Accessors
  double getY();
  void setX(double x);       // Mutators
  void setY(double y);
};
```

- Nun möchten wir eine Kopie eines Punktes erstellen

```
Point p1(3.5, 2.1);
Point p2(p1);                // Copy p1.
```

C++ Copy-Konstruktor (2)

- Dies funktioniert wegen des Copy-Konstruktors
- Der Copy-Konstruktor wird immer dann aufgerufen, wenn eine Kopie eines Objektes erzeugt wird
- Signatur:

```
MyClass(MyClass &other);   // Copy-constructor
```

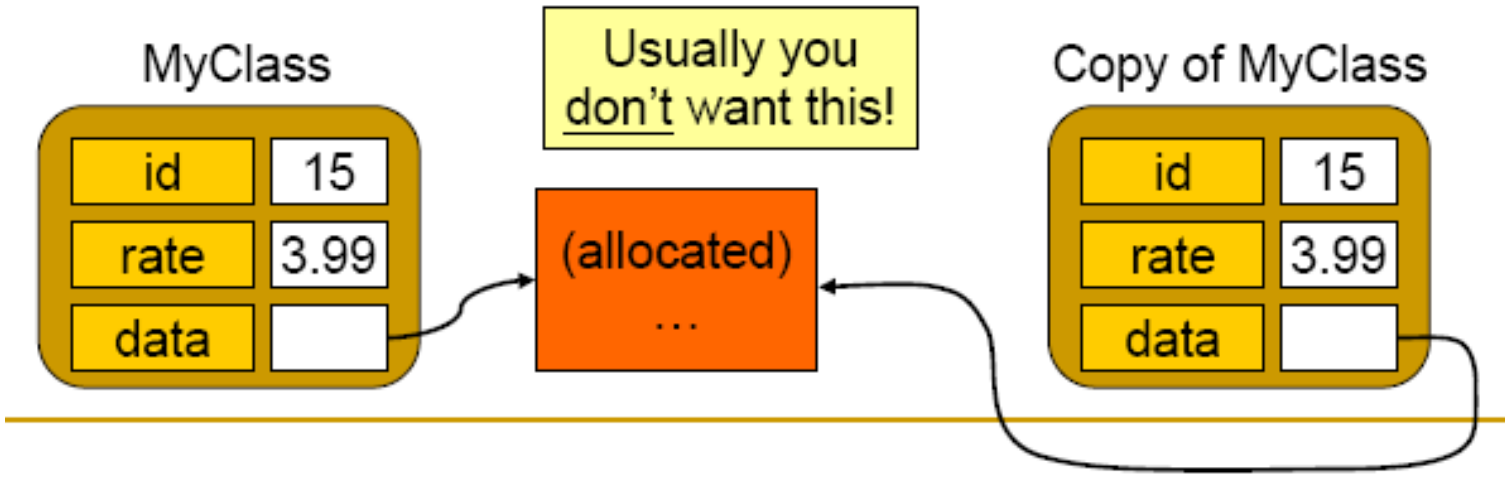
- Argument ist eine Referenz
- Aufrufe sehen z.B. so aus:

```
MyClass(MyClass other);
```

- In der Klasse Point gibt es keinen Copy-Konstruktor
- Der Compiler fügt automatischen einen hinzu
- „Default Copy-Konstruktor“

C++ Copy-Konstruktor (3)

- Der Compiler stellt einen Default Copy-Konstruktor zur Verfügung
- Dieser kopiert einfach alle einfachen Datentypen im Objekt
- Es handelt sich um eine „flache Kopie
  - Wenn im Objekt ein Pointer auf ein Array/Objekt steht, wird nur der Pointer kopiert, nicht das Array/Objekt
  - Das Originalobjekt und die Kopie teilen sich jetzt Speicher!
  - Wenn der Destruktor dynamisch zugeteilten Speicher löscht, führt das zu einem Programmabsturz



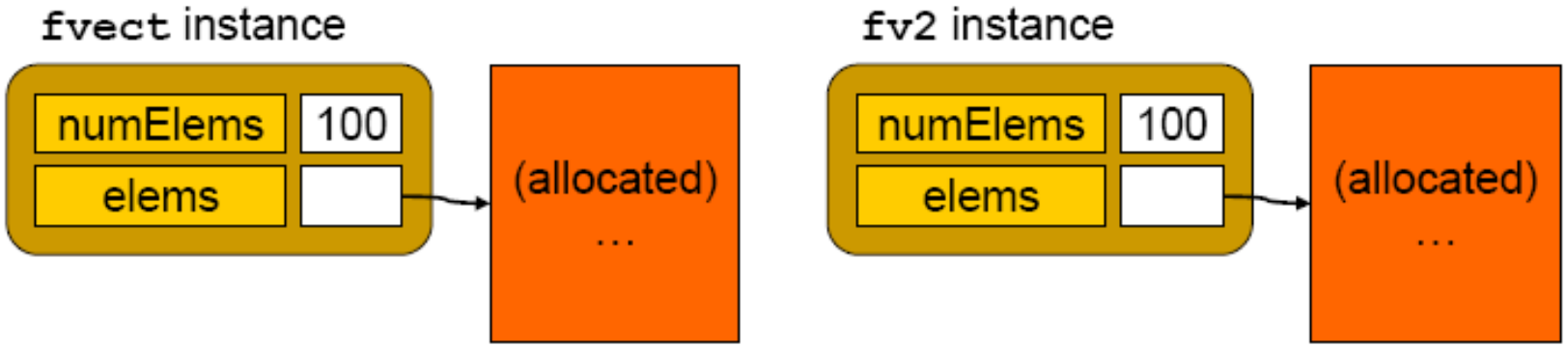
C++ Copy-Konstruktor (4)

- Hier ist der Copy-Konstruktor nicht akzeptabel, da er eine flache Kopie erzeugt

```
FloatVector fv2 = fvect;    // Same as fv2(fvect)
fvect.setelem(3, 5.2);      // Also changes fv2!
```

- Daher muss man seinen eigenen Copy-Konstruktor schreiben:

```
FloatVector::FloatVector(FloatVector &fv) {
  numElems = fv.numElems;
  // DON'T just copy the elems pointer!
  elems = new float[numElems];      // Allocate space
  for (int i = 0; i < numElems; i++) {
    elems[i] = fv.elems[i];         // Copy the data
  }
}
```



C++ Eigenheiten

- C++ setzt bestimmte Klassenoperationen voraus
- Wird etwas nicht gefunden, werden default-Versionen angelegt
- Benötigte Methoden:
  - Ein Copy-Konstruktor
  - Ein Nicht-Copy-Konstruktor
  - Ein Zuweisungsoperator
  - Ein Destruktor
- Aufruf eines Copy-Konstruktors:

```
Point p1(19.6, -3.5);
Point p2(p1);              // Copy p1
```

- Analog (syntaktischer Zucker):

```
Point p1(19.6, -3.5);
Point p2 = p1;             // Identical to p2(p1)
```

- Im Unterschied zu:

```
Point p1(19.6, -3.5), p2;
p2 = p1;
```

C++ Klassen und `structs` (1)

- In C++ sind `structs` wie Klassen
- Sie können Konstruktoren, Member-Funktionen usw. haben
- Der einzige Unterschied ist, dass per default alles `public` ist

```
struct s { ... };
class s { public: ... };     // same thing
```

- Konstruktoren für Klassen sind sinnvoll, z.B. um Initialwerte zu setzen
- `structs` werden in der Regel verwendet, wenn sie nicht den vollen Umfang einer Klasse haben:
- So genannte Helper-Klassen
- „a chunk of Data“
- Das verstecken von `structs` in Klassen ist sinnvoll
  - Teil der Abstrahierung / Kapselung
  - Gutes objektorientiertes Design

C++ Klassen und `structs` (2)

- Man kann `structs` und Klassen in anderen Klassen deklarieren
- Beispiel:

```
class Scheduler {
  private:
  // A "scheduled task" struct
  struct task {
    int id;
    string desc;
    task *next;
  };
  task *schedTasks; // A list of tasks
  ...
};
```

- `task` kann in der Scheduler-Klasse benutzt werden.
- `task` kann nicht in anderen Klassen / außerhalb von Klassen benutzt werden.
- Wenn es `public` wäre, dann als `Scheduler::task`

C++ Klassen und structs (3)

- Verkettete Liste als struct

```
struct node {
  int index;
  int value;
  node *next;   // Pointer to next node in list
};
```

- Besser:

```
struct node {
  int index;    // Index of element in vector
  int value;    // Value of element in vector
  node *next;   // Pointer to next element, or 0
  node(int idx, int val, node *np) :
  index(idx), value(val), next(np) { }
};
```

- Verkettung:

```
node *n1 = new node(3, 5, 0);   // Elem3 = value 5
node *n2 = new node(5, -2, 0);  // Elem5 = value -2
n1->next = n2;
```

Gliederung

1. Einführung in C
2. **Einführung in C++**
   - ...
   - 2.3. Klassen und Objekte
   - 2.4 Dynamische Speicherverwaltung in C++
   - 2.5 Operatoren
   - **2.6 Klassen und Vererbung I**
   - 2.7 IO-Streams
3. C++ für Fortgeschrittene
4. Weitere Themen rund um C++

C++ Klassen und Vererbung (1)

- Beispiel:

```
class Fruit {
  string color;
  int seeds;
public:
  Fruit() : color(""), seeds(0) { }
  Fruit(string col, int s) : color(col), seeds(s) { }
  int getSeeds() const { return seeds; }
  string getColor() const { return color; }
  void print() const {
    cout<< "I am a Fruit!" << endl;
  }
};
```

- Nun erstellen wir verschiedene Instanzen

```
Fruit pear("green", 10);      // pear is a Fruit
Fruit orange("orange", 25);   // orange is a Fruit
```

C++ Klassen und Vererbung (2)

- Der Aufruf der Ausgabe

```
pear.print();
orange.print();
```

- ergibt

```
I am a Fruit!
I am a Fruit!
```

- Nun möchten wir die Eigenschaft repräsentieren, dass die Orange geschält werden kann
- Leider kann `Fruit` das nicht repräsentieren
- Und pears lassen sich nicht schälen
- Lösung: Eine Klasse `Orange`, die die Klasse `Fruit` erweitert!

C++ Klassen und Vererbung (3)

- Die Klasse Orange:

```
class Orange : public Fruit {
  bool peeled; // True if peel has been removed
public:
  Orange() : peeled(false) { }
  bool isPeeled() const { return peeled; }
  void peel() {
    if (peeled) cout << "Already peeled!");
    peeled = true;
  }
};
```

- Orange besitzt zu der Funktionalität der Klasse Fruit neue Funktionalität
- Man kann auch `private` ableiten!

C++ Klassen und Vererbung (4)

- Erweiterung, weil
  - Eine Orange eine Frucht ist
  - Eine Orange die Charakteristik einer Frucht hat
  - Mehr Merkmale, Zustände und Funktionalität hinzugefügt werden
  - Eine Orange eine spezialisierte Version einer Frucht ist

- Neue Terminologie:
  - Fruit ist eine Oberklasse / Super Class / Base Class / Parent Class
  - Orange ist eine Unterklasse / abgeleitete Klasse / Derived Class / Child Class

C++ Klassen und Vererbung (5)

- Anwendungsbeispiel

```
Orange o;
cout << "This orange has " << o.getSeeds()
     << "seeds." << endl;
if (!o.isPeeled()) {
  cout << "Commencing peel removal..." << endl;
  o.peel();    // Peel the orange.
}
```

- Überall, wo wir `Fruit` benutzt haben, können wir jetzt auch `Orange` benutzen!

```
Fruit *pf1 = new Fruit();    // OK
Fruit *pf2 = new Orange();   // Also OK
```

- Eine `Orange` ist also eine `Fruit`
- Eine `Orange` hat die Member, die auch `Fruit` hat

C++ Klassen und Vererbung (6)

- Die Umkehrung gilt nicht!

```
Orange *po1 = new Orange(); // OK
Orange *po2 = new Fruit();  // Compiler error!
```

- Eine `Fruit` ist keine `Orange`
- `Fruit` hat nicht die Member einer `Orange`
- Man kann eine `Fruit` nicht schälen
- Die `Fruit` wurde wie folgt deklariert:

```
class Fruit {
  string color;
  int seeds;  ...
```

- Kann eine Orange auf die Variable `seeds` zugreifen (private)?

```
void Orange::removeSeeds() {
  seeds = 0;
}
```

C++ Klassen und Vererbung (7)

```
void Orange::removeSeeds() {
  seeds = 0;
}
```

- Nein.
- Nur die Klasse selbst kann auf private Elemente zugreifen
- Um Member in Unterklassen zugreifbar zu machen, müssen sie als `protected` definiert sein:

```
class Fruit {
protected: // Make accessible to subclasses!
  string color;
  int seeds;
  ...
}
```

- Nun funktioniert obiges Beispiel

C++ Klassen und Vererbung (8)

- Was sollte `private` sein?
- Was sollte `protected` sein?
- Keine Regel hier, nur Hinweise:
  - In der Oberklasse sollten die Member `private` sein, bis man in einer abgeleiteten Klasse bemerkt, dass man direkt zugreifen muss. Dann sollte man sie auf `protected` abändern.
  - Bei komplizierten Datenstrukturen sollten eher Zugriffsfunktionen verwendet werden, da die abgeleitete Klasse ansonsten viel durcheinander bringen kann.
- Eine abgeleitete Klasse kann Funktionen der Basisklasse überschreiben
- Funktionalität wird ersetzt / spezialisiert

C++ Klassen und Vererbung (9)

- Beispiel:

```
class Orange : public Fruit {
  ...
  void print() const { // Override Fruit::print()
    cout << "I am an Orange!" << endl;
  }
};
```

- Nun ergibt der Aufruf von `print()` einer Instanz von `Orange` folgendes:

```
I am an Orange!
```

C++ Klassen und Vererbung (10)

- Aufruf der `print()`-Funktion der Oberklasse

```
class Orange : public Fruit {
  ...
  void print() const {
    cout << "I am an Orange!" << endl;
    // Call parent-class version of print()
    Fruit::print();
  }
};
```

- Beispiel:

```
Fruit f;
Orange o;

f.print();
o.print();
```

führt zu:

```
I am a Fruit!
I am an Orange!
I am a Fruit!
```

C++ Klassen und Vererbung (11)

- Wie sieht es mit folgendem Code-Fragment aus?

```
void printFruit(const Fruit &fr) {
  fr.print();
}
...
Fruit f;
Orange o;
printFruit(f);
printFruit(o); // An orange is a fruit.
```

- Die Ausgaben sind:

```
I am a Fruit!
I am a Fruit!
```

- Standardmäßig benutzt C++ den Variablentyp, um zu identifizieren, welche Funktion aufgerufen wird
- Nicht den Objekttyp!

C++ Klassen und Vererbung (12)

```
void printFruit(const Fruit &fr) {
  fr.print();
}
```

- `fr` ist vom Type `Fruit`, also wird `Fruit::print()` aufgerufen
- Sogar wenn `fr` eine `Orange` ist!

```
Orange o;
printFruit(o);
```

- `virtual` sagt dem Compiler, dass er prüfen soll, ob die Funktion einer Unterklasse aufgerufen werden muss

C++ Klassen und Vererbung (13)

- Aktualisierung der Klasse Fruit:

```
class Fruit {
  ...
  virtual void print() const {
    cout << "I am a Fruit!" << endl;
  }
};
```

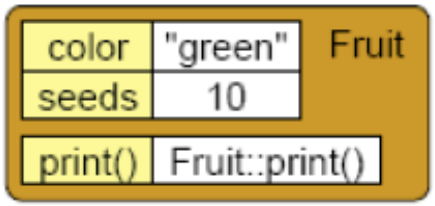
- Dann gibt es die Ausgaben

```
I am a Fruit!
I am an Orange!
```

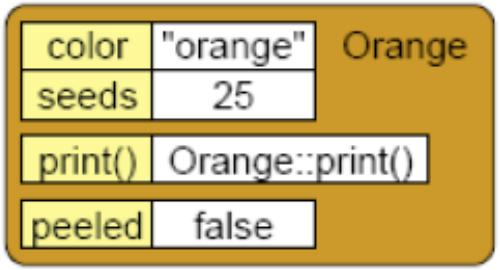
- Durch die Deklaration einer Methode als `virtual` wird der C++-Compiler gezwungen, einen Pointer zu der Instanzfunktion zu speichern
- Damit weiß das Objekt selbst, welche Methode aufgerufen werden muss

C++ Klassen und Vererbung (14)

- Wenn eine `Fruit` erstellt wird, wird `Fruit::print()` als Pointer gespeichert:



- Wenn eine Orange erstellt wird, wird `Orange::print()` gespeichert:



- „Virtual Function Table“
- Benötigte Methoden werden erst zur Laufzeit bestimmt
- Overhead durch Lookup und Dereferenzierung der Funktion-Pointer

C++ Klassen und Vererbung (15)

- Nicht-virtuelle Member können zur Compilezeit bestimmt werden
- Schneller
- Also: Sparsam mit `virtual` sein!
- Wie sieht das mit den Destruktoren aus?

```
Fruit *pf1 = new Fruit();
Fruit *pf2 = new Orange();
...
delete pf1; // Clean up my fruit.
delete pf2;
```

- Beide Varianten rufen `Fruit::~Fruit()` auf!

C++ Klassen und Vererbung (16)

- Der C++-Standard besagt, dass das Löschen einer abgeleiteten Klasse mittels eines Pointers der Basisklasse in einem *undefinierten* Verhalten resultiert!
- Lösung:
- Jede Basisklasse benötigt einen virtuellen Destruktor
- Dann wird schon der richtige Destruktor aufgerufen...

Abstrakte Klassen (1)

- Manche Basisklassen definieren nur Verhalten, sonst nichts
- Die Konzepte einer solchen Basisklasse sind zu allgemein, als dass sie bereits implementiert werden können
- Abstrakte Basisklassen müssen erweitert werden
- Abstrakte Klassen kann man nicht instanzieren
- Implementation der geforderten Funktionalität wird in den Unterklassen definiert
- Eine Klasse ist abstrakt, sobald sie mindestens eine Methode als „pure virtual“ deklariert.

Abstrakte Klassen (2)

- Beispiel:

```
class Figure {
  // ...
public:
  virtual void draw() const {}
};
```

- Besser:

```
class Figure {
  // ...
public:
  virtual void draw() const = 0;
};
```

- Nun müssen wir für jede Figure bestimmen, wie sie gezeichnet wird:

Abstrakte Klassen (3)

```
class Flowchart : public Figure {
    virtual void draw() {
        // Code to draw contends
    }
}
```

- Der Versuch eine `Figure` zu instanzieren führt zu einem Compilerfehler

```
Figure *fig = new Figure();     // Compiler Error
```

- Dennoch kann man `Figure`-Variablen haben:

```
Figure *fig = new Flowchart(); // OK
```

- Basisklassen können einige rein virtuelle Methoden haben

Abstrakte Klassen (4)

- Basisklassen können einige virtuelle Methoden haben
  - Basisklasse stellt dennoch einiges an Implementierungen zur Verfügung
  - Die abgeleitete Klasse ergänzt die fehlenden Teile
- Ein „Interface“ ist eine Basisklasse, die ausschließlich virtuelle Methoden enthält
  - Keine Implementation
  - Nur Verhaltensdefinition
  - Dieses Interface-Konzept ist ausgeprägter in anderen Sprachen