Objektorientiert Programmieren II
Vererbung

Stephan Neuhaus

Lehrstuhl Softwaretechnik
Universität des Saarlandes, Saarbrücken







Letzte Vorlesung

- Was sind Klassen?
- Klassen in C++ und Java
- Methoden und Attribute
- Sichtbarkeit
- Polymorphismus
- Konstruktoren und Destruktoren
- Guter Stil







Ein Beispiel: Verkettete Liste

```
class Item {
public:
  Item(int v, Item *n) { _value = v; _next = n; }
  ~Item() { std::cout << "Deleting item " << _value << std::endl; }
  int value() const { return _value; }
  Item *next() const { return _next; }
private:
  int _value;
  Item *_next;
};
class LinkedList {
public:
  LinkedList() { items = 0; _length = 0; }
  ~LinkedList();
  void push(int i);
  int pop(int *i);
  int top(int *i) const;
  int length() const { return _length; }
private:
  Item *items;  int _length;
};
```







Verkettete Liste: Destruktor

Der Destruktor geht durch die Liste und löscht jedes einzelne Element. Da muß man etwas vorsichtig zu Werke gehen, weil nach einer `delete`-Operation das Objekt per Definition tot ist. Will man bestimmte Felder nach `delete` weiterverwenden, muß man sie vorher retten.

```
LinkedList::~LinkedList() {
  Item *nextItem;

  for (Item *i = items; i != 0; i = nextItem) {
    nextItem = i->next();
    delete i;
  }
}
```







Verkettete Liste: push() *und* top()

Die Aufrufschnittstelle zu top() muß sowohl den Wert des obersten Elements zurückgeben als auch einen Indikator, ob die Liste leer war.

```
void LinkedList::push(int i) {
  Item *item = new Item(i, items);
  items = item;
  _length++;
}

int LinkedList::top(int *i) const {
  if (items != 0) {
    *i = items->value();
    return 1;
  } else
    return 0;
}
```







Verkette Liste: pop

Eindeutig die komplizierteste Methode. Sie sorgt dafür, daß die Liste nach der Operation immer noch korrekt verkettet ist.

```
int LinkedList::pop(int *i) {
  if (items != 0) {
    Item *top = items;
    items = top->next();
    *i = top->value();
    delete top;
    _length--;
    return 1;
  } else
    return 0;
}
```

Verkettete Liste: Benutzung

```
int main (int argc, const char *argv[]) {
  LinkedList l;

  l.push(1); l.push(4); l.push(9); l.push(16);

  int i;
  while (l.top(&i)) {
    std::cout << i << " " << l.length() << "; ";
    l.pop(&i);
  }
  std::cout << "Length: " << l.length()
            << " (should be 0)" << std::endl;

  l.push(2); l.push(3); l.push(5); l.push(7);
  std::cout << "Length: " << l.length()
            << " (should be 4)" << std::endl;
  return 0;
}
```

Verkettete Liste: Ausführung

```
% g++ -o LL1 LL1.cc
% ./LL1
16 4; Deleting item 16
9 3; Deleting item 9
4 2; Deleting item 4
1 1; Deleting item 1
Length: 0 (should be 0)
Length: 4 (should be 4)
Deleting item 7
Deleting item 5
Deleting item 3
Deleting item 2
```







Kleine Tricks I

Wir wollen `l.push(1).push(4).push(9)` schreiben können.

```
class LinkedList {
public:
  // ...
  LinkedList &push(int i);
  // ...
};

LinkedList &LinkedList::push(int i) {
  Item *item = new Item(i, items);
  items = item;
  _length++;
  return *this;
}

int main (int argc, const char *argv[]) {
  LinkedList l;
  l.push(1).push(4).push(9).push(16);
}
```







Vererbung

Wenn nach der öffnenden Klammer einer `class`-Deklaration nichts steht, wird `private` angenommen

```
class Point { /* ... */ }
class Color { /* ... */ }
enum Kind { circle, triangle, square };

class Shape {
  Kind k;
  Point center;
  Color color;
public:
  void draw() const;
  void rotate(int);
};
```







Shape-*Klasse: Implementierung*

Da man verschiedene Arten von Formen haben kann, muß für jede dieser Arten eine eigene draw_xxx()-Routine implementiert werden

```
void Shape::draw(int angle) const {
  switch (k) {
  case circle: draw_circle(); break;
  case triangle: draw_triangle(); break;
  case square: draw_square(); break;
  }
}
```

Will man mehr Formen hinzufügen, muß man an verschiedenen Stellen das Programm ändern und neu kompilieren

Hier helfen Ableitungen







Ableitungen

Ableitungen werden verwendet, um Spezialisierungen zu modellieren. Diese Spezialisierungen heißen auch „ist-ein“-Beziehungen (engl. is-a relationship)

Ein Auto ist-ein Fahrzeug, ein Dreieck ist-ein Vieleck usw.

Ist-ein Beziehungen sind *gerichtet*







Ableitungen: Syntax I

Memberinitialisierungen mit : member(wert) im Konstruktor

```
#include <string>
#include <iostream>

class Base {
public:
  std::string name;
  Base(std::string _name) : name(_name) {
    std::cout << "Base's constructor for " << name << std::endl;
  }
  ~Base() {
    std::cout << "Base's destructor for " << name << std::endl;
  }
  void method() {
    std::cout << "Base's method for " << name << std::endl;
  }
};
```







Ableitungen: Syntax II

Initialisierung von Basisklassen mit
`: Basisklasse(Parameter)` im Konstruktor

```
class Derived : public Base {
public:
  Derived(std::string name) : Base(name) {
    std::cout << "Derived's constructor for "
              << name << std::endl;
  }
  ~Derived() {
    std::cout << "Derived's destructor for " << name << std::endl;
  }
  void method() {
    std::cout << "Derived's method for " << name << std::endl;
  }
};
```







Ableitungen: Syntax III

Jedes `Derived`-Objekt ist-ein `Base`-Objekt

Typumwandlung über Referenz, nicht über Objekt, weil sonst temporäres Objekt erzeugt und wieder zerstört werden könnte

```
int main(int argc, const char *argv[]) {
  Base base("base");
  base.method();

  Derived derived("derived");
  derived.method();

  Base &baseRef = derived;        // No cast necessary
  Derived &derivedRef = derived;

  derivedRef.method();
  baseRef.method();
  return 0;
}
```







Ableitung: Output

Die Konstruktoren werden in der Reihenfolge Base, Derived aufgerufen, die Destruktoren umgekehrt

```
Base's constructor for base          # Base base("base");
Base's method for base               # base.method();
Base's constructor for derived       # Derived derived("derived");
Derived's constructor for derived    # Derived derived("derived");
Derived's method for derived         # derived.method();
Derived's method for derived         # derivedRef.method();
Base's method for derived            # baseRef.method();
Derived's destructor for derived     # return 0, ~Derived, derived
Base's destructor for derived        # return 0, ~Base, derived
Base's destructor for base           # return 0, ~Base, base
```

Sichtbarkeit: Attribute und Methoden

Hat man Klasse B von Klasse A abgeleitet, sagt man „B erbt von A“. Die Klasse B erbt insbesondere diie Attribute und Methoden.

- Bei `public` inheritance haben alle `public` Methoden und Attribute von A auch in B die Sichtbarkeit `public`
- Bei `private` inheritance haben alle `public` Methoden und Attribute von A auch in B die Sichtbarkeit `private`

Es gibt noch `protected` inheritance, aber wir lassen das erst mal weg.

In der Paxis eigentlich immer `public` inheritance

Sichtbarkeit: `public` *inheritance*

```
class Base {
public:
  void f() {}
  void g() {}
private:
  void h() {}
};

class Derived : public Base {
public:
  void f() { Base::h(); } // Error: can't access Base::h()
  void h() { g(); }       // OK, calls Base::g()
};

void use() {
  Derived d;
  d.f();       // Calls Derived::f() on d
  d.g();       // Calls Base::g() on Base part of d
  d.Base::f(); // Calls Base::f() on Base part of d
}
```







Sichtbarkeit: `private` *inheritance*

Private inheritance wird gebraucht, um Code-Duplizierung zu vermeiden, nicht, um gemeinsame Interfaces zu implementieren

```
class Base {
public:
  void f() {}
};
class Derived : private Base {
public:
  void g() { Base::f(); } // OK: access Base's public members in Derived
};

void use() {
  Derived d;

  d.f(); // Error: Can't access Base's public members through Derived
  d.g(); // OK
}
```







protected *Attribute*

Attribute, die in abgeleiteten Klassen, aber nicht von außen sichtbar sein sollen, werden als `protected` gekennzeichnet

```
class Base {
public: int x;
protected: int y;
private: int z;
};

class Derived : public Base {
public:
  void f() { z = 1; } // Error: z is private
  void g() { y = 1; } // OK: y is protected
}

void h(Derived* d) {
  d->x = 1; // OK: x is public
  d->y = 1; // Error: y is protected
}
```

Obacht!

Die Sichtbarkeitsregeln ergeben einen Schutz des Programmierers, keine Sicherheitsmaßnahmen

Sichtbarkeitsregeln werden zur Kompilierzeit geprüft, nicht zur Laufzeit!

Sichtbarkeitsregeln können (mit Geduld) immer umgangen werden

Wenn man das häufig macht, ist in der Regel aber irgend etwas anderes kaputt







Virtuelle Methoden

Ableitung der Klassen `Triangle`, `Circle`, `Square` usw. von Shape möglich. Angenommen, alle Ableitungen implementieren `draw()`:

```
void Shape::draw(int angle) const {
  switch (k) {
  case circle: ((Circle *)this)->draw(); break;
  case triangle: ((Triangle *)this)->draw(); break;
  case square: ((Square *)this)->draw(); break;
  }
}
```

Nicht so toll!

Gesucht: Möglichkeit, aus einer Basisklasse Methoden in der abgeleiteten Klasse mit gemeinsamem Namen aufzurufen







Virtuelle Methoden

```
#include <iostream>

class Shape {
public: virtual void draw() const = 0;
};

class Triangle : public Shape {
public: void draw() const {
    std::cout << "Circle's draw() called" << std::endl;
  }
};

// Another possibility: void f(const Shape *s) { s->draw(); }
void f(const Shape &s) { s.draw(); }

int main(int argc, const char *argv[]) {
  Triangle t;
  f(t);        // Another possibility: f(&t);
}
```







Default-Implementierung

```
#include <iostream>

class Shape {
public: virtual void midpoint() const {
    std::cout << "Shape's midpoint called" << std::endl;
  }
};

class Triangle : public Shape {
public: void midpoint() const {
    std::cout << "Triangle's midpoint called" << std::endl;
  }
};
class Circle : public Shape { };

void f(const Shape &s) { s.midpoint(); }

int main(int argc, const char *argv[]) {
  Triangle t; f(t); // Calls Triangle::midpoint()
  Circle c; f(c);   // Calls Shape::midpoint()
}
```

Vererbung in Java

```
class A {
    public void f() { System.out.println("A.f()"); }
}

class B extends A {
    public void g() { System.out.println("B.g()"); }
}

public class Main1 {
  public static void main(String[] args) {
      A a = new A();
      B b = new B();
      a.f();
      b.g();
      b.f();
  }
}
```







Kompilierung und Ausführung

```
% javac Main1.java
% java -classpath . Main1
A.f()
B.g()
A.f()
```







Virtuelle Methoden in Java I

Methoden sind in Java automatisch `virtual`

```
// File Main2.java
class A {
    public void f() { System.out.println("A.f()"); }
}
class B extends A {
    public void g() { System.out.println("B.g()"); }
    public void f() { System.out.println("B.f()"); }
}

public class Main2 {
    private static void foo(A a) {
        a.f();
    }
    public static void main(String[] args) {
        A a = new A(); foo(a); // Calls A.f()
        B b = new B(); foo(b); // Calls B.f()
    }
}
```







Virtuelle Methoden in Java II

Methoden ohne Default-Implementierung heißen abstract

```
// File Main3.java
abstract class A { // Class with all methods abstract is abstract
    abstract public void f();
}

// Error: B does not define A.f() (define f or make B abstract)
class B extends A {
}
// OK: defines abstract method f()
class C extends A {
    public void f() { }
}

public class Main3 {
    public static void main(String[] args) {
        A a = new A(); // Error: abstract A cannot be instantiated
    }
}
```







Virtuelle Destruktoren: So nicht!

```
class Base { // Bug! Don’t do this!
public:
  virtual void show() = 0;
};

class Derived {
  int* array;
public:
  Base() { array = new int[100]; }
  ~Base() { delete array; }
  void show() { cout << "Hi" << endl; }
};

void f(Base *b) {
  delete b;
}

void g() {
  f(new Derived());
}
```







Virtuelle Destruktoren: Lieber so!

```
class Base {
public:
  virtual void show() = 0;
  virtual ~Base() { } // Note virtual destructor
};

class Derived {
  int* array;
public:
  Base() { array = new int[100]; }
  ~Base() { delete array; }
  void show() { cout << "Hi" << endl; }
};

void f(Base *b) {
  delete b;
}

void g() {
  f(new Derived());
}
```







Virtuelle Konstruktoren I

Der Copy-Konstruktor kann bei Klassenhierarchien nicht das „ganze“ Objekt kopieren (muß den *genauen* Typ kennen):

```
class Base {
public:
  Base(const Base &c) { x = c.x; }
private:
  int x;
};
class Derived : public Base {
public:
  Derived(const Derived &c) : Base(c) { y = c.y; }
private:
  int y;
};

int f(Derived* p) {
  Base* q = new Base(p);    // Copies only Base part of *p
  Base* r = new Derived(p); // Copies also Derived part of *p
}
```







Virtuelle Konstruktoren II

```
class Base {
  int x;
public:
  Base(const Base& c) { x = c.x }
  virtual Base* clone() { return new Base(*this); }
};

class Derived : public Base {
  int y;
public:
  Base(const Base& c) { x = c.x }
  Derived* clone() { return new Derived(*this); }
};

int f(Derived* p, Base* q) {
  Derived* r = p->clone();
  Base* s = p->clone();
  Derived* t = q->clone();                  // Error
  Derived* u = dynamic_cast<Derived*>(s);  // OK
}
```







Interfaces

Interfaces existieren in Java, um klarzustellen, daß ein Objekt bestimmte Methoden mit einer bestimmten Signatur anbieten muß

Interfaces existieren nicht als solche in C++. Man kann sie jedoch durch abstrakte Klassen und Mehrfachvererbung emulieren







Interfaces in Java

```
public interface Patient {
  public Gesundheitsdaten ermittleGesundheitsdaten();
}

public class Person implements Patient {
  public Gesundheitsdaten ermittleGesundheitsdaten() {
    // ...
  }
}
```

Wichtige Interfaces: `Serializable`, `Comparable`

Sie können mehrere Interfaces gleichzeitig implementieren







Interfaces in C++

Interfaces werden in C++ nicht angeboten

Sie müssen sich also mit anderen Mechanismen behelfen

```
class Patient {
public:
  virtual const HealthData &getHealthData()  const = 0;
  virtual ~Patient() {}; // Virtual Destructor!
};

class Person : public Patient {
public:
  const HealthData &getHealthData() const {
    // ...
  }
  ~Person() { /* ... */ }
};
```

Mehr als ein Interface implementieren: Mehrfachvererbung

Mehrfachvererbung

Mehrfachvererbung: Man erbt von mehreren Oberklassen gleichzeitig

Mehrfachvererbung ist ein *sehr* trickreiches Kapitel von C++

Es gibt sehr viele Fallstricke und sehr viele Subtilitäten zu beachten

In Java ist man da besser raus, weil man gar nicht in die Verlegenheit kommt (In C++ ist das erlaubt, also wird's auch gemacht)

Am besten nur in eng begrenzten Anwendungsdomänen (z.B. Interfaces) anwenden

Mehrfachvererbung: Beispiel

```
class GraphicObject {
public:
  void draw() { /* ... */ }
};

class Shape {
public:
  double area() { /* ... */ }
};

class VisualRectangle : public Shape, public GraphicObject {
};

void f(VisualRectangle *p) {
  p->draw();              // Calls GraphicObject::draw()
  double d = p->area();  // Calls Shape::area();
}
```







Mehrfachvererbung: Disambiguation(!)

Disambiguation: Entmehrdeutigung? Einfach Auflösen von Mehrdeutigkeiten

```
class GraphicObject {
public:
  void draw() { /* ... */ }
};
class Shape {
public:
  void draw() { }
};

class VisualRectangle : public Shape, public GraphicObject {
};

void f(VisualRectangle *p) {
  p->draw();                 // Error: ambiguous
  p->Shape::draw();          // Calls Shape::draw()
  p->GraphicObject::draw(); // Calls GraphicObject::draw()
}
```

Disambiguation durch Überladen

```
class GraphicObject {
public:
  void draw() { /* ... */ }
};
class Shape {
public:
  void draw() { }
};

class VisualRectangle : public Shape, public GraphicObject {
public:
  void draw() { // Means both Shape::draw() and GraphicObject::draw()
    GraphicObject::draw();
    Shape::draw();
  }
};

void f(VisualRectangle *p) {
  p->draw();                 // OK, no longer ambiguous
}
```







Virtuelle Funktionen gleichen Namens

```
class GraphicObject {
public:
  virtual void draw() { /* ... */ }
};
class Shape {
public:
  virtual void draw() { }
};
class VisualRectangle : public Shape, public GraphicObject {
public:
  // Overrides both Shape::draw() and GraphicObject::draw()
  void draw() {
    GraphicObject::draw();
    Shape::draw();
  }
};

void f(VisualRectangle *p) {
  p->draw();                 // OK, no longer ambiguous
}
```







Replizierte Basisklassen

Grundsätzlich ist es mit Mehrfachvererbung möglich, zweimal dieselbe Basisklasse in die Vererbungshierarchie hineinzubekommen

```
struct Link {
  Link* next;
};

class Task : public Link {
  // Use link to link different tasks
};

class Displayed : public Link {
  // Use link to maintain list of displayable elements
};

class Satellite : public Displayed, public Task {
  // ...
};
```

Replizierte Basisklassen

```
void f(Satellite* p) {
  p->next = 0;             // Error: ambiguous: which Link?
  p->Link::next = 0;       // Error: ambiguous: which Link?
  p->Task::next = 0;       // OK
  p->Displayed::next = 0; // OK
}
```

Ein `Satellite` enthält zwei Objekte vom Typ `Link`

Wenn das nicht geht, muß man virtuelle Basisklassen nehmen







Virtuelle Basisklassen

```
struct Link {
  Link* next;
};

class Task : public virtual Link {
  // Use link to link different tasks
};

class Displayed : public virtual Link {
  // Use link to maintain list of displayable elements
};

class Satellite : public Displayed, public Task {
  // ...
};
```

Enthält nur ein Objekt vom Typ `Link`







Guter Stil

- Vermeiden Sie Mehrfachvererbung
- Wenn Sie Mehrfachvererbung verwenden müssen, stellen Sie sicher, daß Sie die Regeln kennen
- Benutzen Sie Mehrfachvererbung, um Interfaces zu beschreiben
- Wenn Sie Interfaces als Sprachmittel zur Verfügung haben, nutzen Sie die statt Mehrfachvererbung (toller Tip, Sie haben in der Regel entweder Interfaces oder Mehrfachvererbung, aber nicht beides!)
- Benutzen Sie Vererbung, um Attribute oder Methoden in gemeinsame Basisklassen auslagern