Programmieren in C++

Vererbung

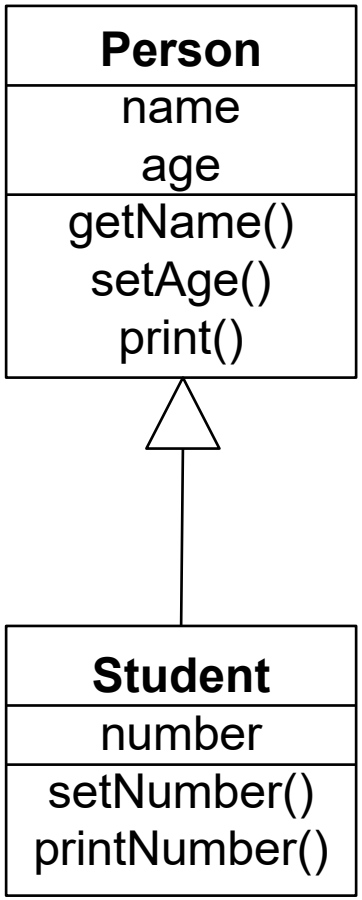
Inhalt

- Vererbung
- Konstruktoren und Destruktor in abgeleiteten Klassen
- Typkonvertierung von Zeigern auf Klassen
- RTTI
- Verdecken und Überschreiben
- Polymorphie
- Überschreiben von Methoden
- Vererbung bzw. Überschreiben unterbinden
- Zugriffsrechte
- Polymorphie von automatisch erstellten Methoden erzwingen
- Mehrfachvererbung

Vererbung: Beispiel Person/Student

- Superklasse
  (Basisklasse, Oberklasse, Vaterklasse)
  Klasse Person mit Eigenschaften
  - Datenfelder: name, age
  - Methoden: getName(), setAge(), print()

- Subklasse
  (abgeleitete Klasse, Unterklasse, Sohnklasse)
  - erbt Eigenschaften der Superklasse
  - fügt eigene Eigenschaften hinzu
    - number
  - fügt neue Methoden hinzu
    - setNumber()
    - printNumber()



Realisierung der Klasse Person

```
// in h-Datei
class Person {
    string m_name;              // Aggregation: Person hat einen Namen
    int m_age;                  // Aggregation: Person hat ein Alter

public:
    Person(const char name[], int age) : m_name(name), m_age(age) {}
    string getName() const   { return m_name; }
    void setAge(int age)     { m_age = age; }
    void print() const;         // keine inline-Implementierung
};

// in cpp-Datei
void Person::print() const {
    cout << "Name: " << m_name << endl;
    cout << "Alter: "   << m_age << endl;
}
```

Realisierung der Klasse Student

```
// in h-Datei: Vererbung: ein Student ist eine Person
class Student : public Person {
    // die Klasse Student wird von der Klasse Person abgeleitet
    // und erbt alle Attribute und Methoden der Klasse Person
    int m_number;

public:
    Student(const string& name, int age, int nr)
        : Person(name, age), m_number(nr) {}
    // neue Methoden der Klasse Student
    void setNumber(int nr) { m_number = nr; }
    void printNumber() const;
};

// in cpp-Datei
void Student::printNumber() const {
    cout << "Studentennummer: " << m_number << endl;
}
```

Verwendung der Klasse Student

```
void main () {
    Person pers("Peter", 20);
    pers.setAge(21);
    pers.print();

    Student student("Anna", 21, 50101);
    student.setName("Anne");
    student.setNumber(56123);
    student.print();                    // gibt keine Studentennummer aus
    student.printNumber();              // gibt Studentennummer aus

    Person pers2 = student;             // Projektion von Student auf Person (Kopie)
    pers2.print();                      // gibt keine Studentennummer aus
}
```

Konstruktoren in abgeleiteten Klassen

- Idee
  - jeder abgeleitete Konstruktor initialisiert nur die neuen Attribute
  - vererbte Attribute werden vom Konstruktor der Basisklasse initialisiert
- Umsetzung
  - In der Initialisierungsliste des Konstruktors wird der Konstruktor der Basisklasse aufgerufen
  - falls kein expliziter Aufruf eines Konstruktors der Basisklasse erfolgt, wird der Standardkonstruktor der Basisklasse implizit aufgerufen
  - Aufgaben der Initialisierungsliste (Reihenfolge beachten)
    - Aufrufen von Konstruktoren der Basisklasse(n)
    - Aufrufen von anderen Konstruktoren der eigenen Klasse (Constructor delegation) oder Initialisieren der eigenen Attribute

Konstruktoren erben

- Grundsatz
  - normalerweise erbt eine Klasse keine Konstruktoren ihrer Basisklassen

  → somit stehen nur die eigenen Konstruktoren zur Erzeugung von Instanzen zur Verfügung

- mittels using können alle Konstruktoren der Basisklasse geerbt werden

```
class Student: public Person {
      int m_matNr = 0;                          // Initialwert ist hier wichtig
public:
      using Person::Person;                     // erbt alle Konstruktoren der Klasse Person
      Student(const string& name, const string& vname, int m)
      : Person(name, vname), m_matNr(m) {
            cout << "ctor von Student" << endl;
      }
}
```

  - Das System verwendet den geerbten Konstruktor der Klasse Person und erzeugt einen Konstruktor für die Klasse Student mit der gleichen Deklaration.
  - Der erzeugte Konstruktor der Klasse Student ruft den Konstruktor der Klasse Person mit den gleichen Argumenten auf.

Destruktor einer abgeleiteten Klasse

- Konzept
  - der Destruktor einer abgeleiteten Klasse ruft nach Ausführung seines Methodenkörpers den Destruktor der Basisklasse implizit auf
  - dynamische Attribute können im Destruktor zuerst gelöscht werden, bevor Attribute der Basisklasse gelöscht werden
- Wann soll ein Destruktor ausprogrammiert werden?
  - wenn die Klasse Attribute enthält, welche eigenständig mit new erzeugt worden sind, so müssen diese im Destruktor wieder gelöscht werden
- Wird der Destruktor auch bei einem statisch erzeugten Objekt aufgerufen?
  - Ja! Beim Verlassen des Blocks, in dem das Objekt erstellt worden ist, wird zuerst der Destruktor aufgerufen, bevor das Objekt vom Stack entfernt wird.

Overload Resolution

- Typisches Szenario
  - Basisklasse und abgeleitete Klasse enthalten beide eine Methode foo mit leicht unterschiedlicher Signatur

    ```
    void Vater::foo(char)
    void Sohn::foo(int)
    ```

  - Welche Methode wird aufgerufen?

    ```
    Sohn s;
    s.foo(5);          // Sohn::foo wird aufgerufen, ok!
    s.foo('A');        // Sohn::foo wird aufgerufen, weshalb?
    ```

  - Wie kann foo von Vater aufgerufen werden?
    - wenn die Sohn-Klasse die Methode foo ihres Vaters anbietet: using Vater::foo;
    - oder explizit aufrufen: s.Vater::foo('A');

Typkonvertierungen von Zeigern

- Typ einer Zeiger- oder Referenzvariable muss nicht gleich dem Typ des Objektes sein, auf welches die Zeiger-/Referenzvariable verweist
  - bisher: `Student *pStud = new Student("Anna", 21, 50101);`
  - neu: `Person *pPers = new Student("Anna", 21, 50101);`

- implizite (automatische) Zeigertypkonvertierung (Up-Cast)

```
Person *pPers2 = pStud;                                  // impliziter Up-Cast
```

- explizite Zeigertypkonvertierung (Down-Cast)

```
Student *pStud2 = dynamic_cast<Student*>(pPers); // expliziter Down-Cast
```

Gültige Up- und Down-Casts

- Up-Cast
  - Konvertierung in einen Zieltyp, der in der Vererbungshierarchie weiter oben liegt
  - implizite Konvertierung
  - immer gültig, wenn der Zieltyp ein Vorfahre ist
- Down-Cast
  - Konvertierung in einen Zieltyp, der in der Vererbungshierarchie weiter unten liegt
  - nur explizite Konvertierung möglich
  - nur gültig, wenn der Zeiger auf ein Objekt des Zieltyps oder einer abgeleiteten Klasse des Zieltyps zeigt
- Beispiele

```
Person *pPers = new Person();
Student *pStud = new Student();
Person *pS = pStud;                          // impliziter Up-Cast
Student *pS2 = static_cast<Student*>(pS);    // gültiger Down-Cast
Student *pS3 = static_cast<Student*>(pPers); // ungültiger Down-Cast
```

Runtime Type Information (RTTI)

- Problem
  - static_cast oder C-Cast führen bei ungültigem Down-Cast zu Laufzeitfehlern
- RTTI
  - speichert genauen Typ zu jeder Instanz
  - kann bei Bedarf abgeschaltet werden
- dynamic_cast
  - bei einem gültigen Down-Cast
    - funktioniert wie ein static_cast
    - `Student *pS4 = dynamic_cast<Student*>(pS);` `// pS4 == pS`
  - bei einem ungültigen Down-Cast
    - gibt einen nullptr zurück (bei einer Zeigervariablen)
    - `Student *pS5 = dynamic_cast<Student*>(pPers);` `// pS5 == nullptr`
    - wirft bad_cast Exception (bei einer Referenzvariablen)

Typkonvertierung mit Smart-Pointers

- Funktioniert analog zu Zeigern

```
shared_ptr<Person> spP = make_shared<Person>();
shared_ptr<Person> spS = make_shared<Student>();
```

- gültige Down-Casts

```
auto sp1 = static_pointer_cast<Student>(spS);
auto sp2 = dynamic_pointer_cast<Student>(spS);
```

- ungültiger Down-Cast

```
auto sp3 = dynamic_pointer_cast<Student>(spP); // sp3 == nullptr
```

Typinformation bei RTTI

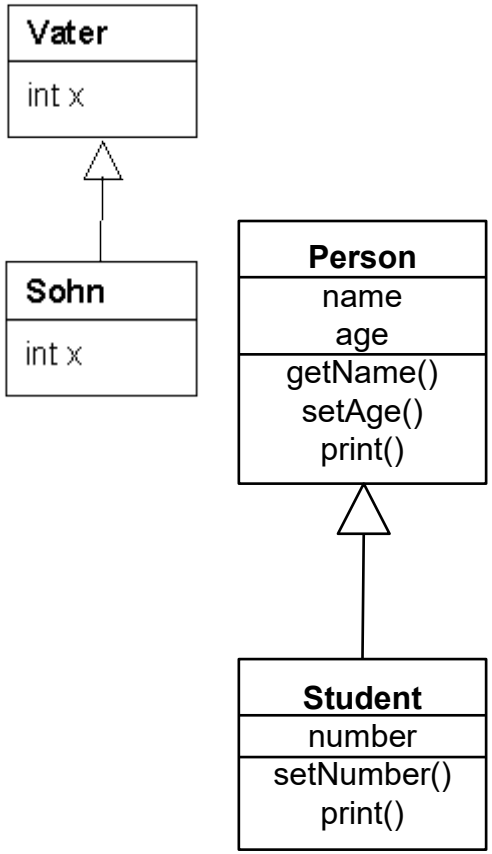
- Operator typeid
  - Syntax: *type_info& t* = typeid(* *Zeigervariable*);
  - gibt Referenz auf Typ-Informationsobjekt zurück
  - benötigt #include <typeinfo>
- Beispiel

```
Person *pPers = new Person();
Person *pStud = new Student();
const type_info& tP = typeid(*pPers);
const type_info& tS = typeid(*pStud);
if (tP == tS) cout << "beide Typen sind gleich" << endl;
if (tP.before(tS)) cout << "tP ist eine Basisklasse von tS" << endl;
```

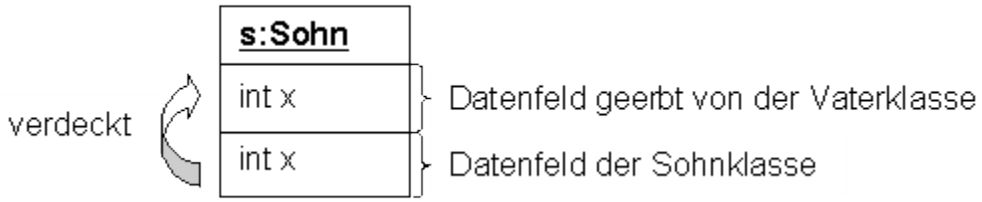
Verdecken und Überschreiben

- Verdecken
  - abgeleitete Klassen können Datenfelder enthalten, die den gleichen Namen haben wie Datenfelder in den Basisklassen, z.B. x
  - sollte wenn möglich vermieden werden

- Überschreiben (overriding)
  - abgeleitete Klassen können Methoden enthalten, die die genau gleichen Signaturen haben wie Methoden in den Basisklassen, z.B. print()
  - wird sinnvoll eingesetzt



Verwendung verdeckter Attribute




```
class Vater {
   protected: int x;
};
class Sohn: public Vater {
   int x;
public:
   Sohn(int xx) : x(xx) {
      cout << "x des Sohnes: " << x << endl;
      cout << "x des Sohnes: " << this->x << endl;
      cout << "vom Vater geerbtes x: " << Vater::x << endl;
      cout << "vom Vater geerbtes x: " << static_cast<Vater *>(this)->x<< endl;
   }
};
```

Polymorphie (Vielgestaltigkeit)

- Polymorphie von Operationen
  - gleiche Methodenaufrufe in verschiedenen Klassen führen zu klassenspezifischen Anweisungsfolgen
  - Beispiel: pPers->print() vs. pStud->print()

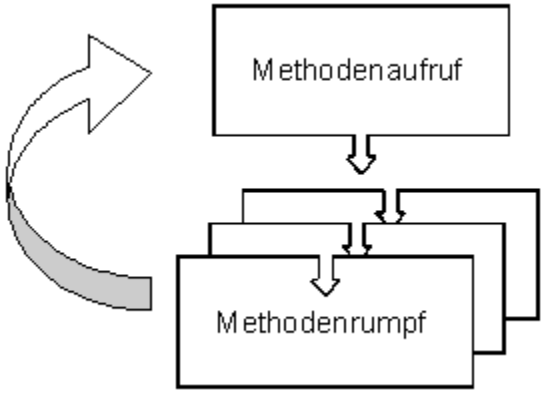
- Polymorphie von Objekten (nur bei Vererbungshierarchien)
  - an die Stelle eines Objektes in einem Programm kann auch ein Objekt einer abgeleiteten Klasse treten
  - ein abgeleitetes Objekt ist polymorph: es kann sich auch als Objekt einer Basisklasse ausgeben
  - Beispiel: ein Student verhält sich wie ein Student, kann sich aber auch wie eine Person verhalten

Statische und dynamische Bindung

- Bindung

  Zuordnung eines Methodenrumpfes zum Aufruf einer Methode

- statische (frühe) Bindung
  - Zuordnung erfolgt zur Kompilationszeit
  - erlaubt Methodenaufrufe durch Methodencode zu ersetzen
  - Standardverhalten
- dynamische (späte) Bindung
  - Zuordnung erfolgt erst zur Laufzeit des Programms
  - sehr mächtiges Konzept, weil es die Wiederverwendung von Programmcode drastisch erhöht
  - muss explizit mit dem Schlüsselwort virtual deklariert werden
  - benötigt pro Objekt einen versteckten Zeiger auf eine Tabelle (vtable) mit den dynamisch gebundenen Methoden



Überschreiben von Methoden

- Idee
  - in einer abgeleiteten Klasse kann eine Methode überschrieben (override) werden
  - die überschriebene Methode hat
    - die gleiche Signatur (Name und Parameterliste)
    - und den gleichen Rückgabetyp oder bei Referenz-/Zeigertyp auch eine Spezialisierung davon
  - wird eine Methode in einer Basisklasse als virtual deklariert, so sind auch alle überschriebenen Methoden davon virtual
- Beispiel

```
class Person { … virtual void print() const { … } … };
class Student : public Person {
    void print() const override {    // die Methode soll explizit als
        …                            // überschrieben gekennzeichnet werden
        Person::print();   // Aufruf der Methode print() aus der Basisklasse
    }
};
```

Gebundene Methoden

- Falls Methoden nicht virtual sind: statische Bindung
  - der statische Typ des Objekts, Zeigers oder Referenz entscheidet über die Wahl der aufgerufenen Methode
- Falls Methoden virtual sind: dynamische Bindung
  - Zugriff über Zeiger/Referenz: Polymorphie kommt zum Einsatz
    - und der dynamische Typ des Zeigers oder der Referenz entscheidet über die Wahl der aufgerufenen Methode
  - direkter Zugriff: Polymorphie kommt nicht zum Einsatz
    - weil die Methode nicht über einen Zeiger bzw. Referenz aufgerufen wird
  - Beispiele

```
Person p, *pP;
Student s, *pS = new Student();
p = s; p.print();              // print() der Klasse Person wird aufgerufen
pP = pS; pP->print();          // print() der Klasse Student wird aufgerufen
Person& rP = s; rP.print();    // print() der Klasse Student wird aufgerufen
```

Vererbung unterbinden

- Vererbung einer Klasse verunmöglichen
  - mit final markierte Klasse kann nicht abgeleitet werden
  - Beispiel

```
class B { … };
class C final : B { … };
class D : C { … };      // führt zu einer entsprechenden Fehlermeldung
```

- Überschreiben von Methoden verunmöglichen
  - mit final markierte Methode darf in abgeleiteter Klasse nicht überschrieben werden
  - Beispiel

```
struct B { virtual void f(int) {} };
struct C : B {
      void f(int) final override {}
};
struct D : C {
     void f(char) {}      // neue Methode, weil andere Signatur
     void f(int) {}       // führt zu einer entsprechenden Fehlermeldung
};
```

Zugriffsrechte

Zugriffsrechte der Basisklasse	**Basisklasse geerbt als**	**Zugriffsrechte Bei der Benutzung der abgeleiteten Klasse**
public protected private	public	public protected no access[1]
public protected private	protected	protected protected no access[1]
public protected private	private	private private no access[1]

[1] Ausser friend-Deklaration in Basisklasse erlaubt den Zugriff explizit.

Interfaces

- Interface
  - es existiert kein «Interface»-Typ
  - abstrakte Klasse ohne Implementierung entspricht einem Interface
- abstrakte Methoden und Klassen
  - eine virtuelle Methode ohne Implementierung ist abstrakt
  - eine Klasse mit mindestens einer abstrakten Methode ist selber abstrakt
  - von abstrakten Klassen können keine Instanzen erzeugt werden
- Beispiel

```
struct VehicleInterface {
    virtual ~Vehicle() = default;
    virtual void drive() = 0; // abstrakt (pure virtual)
};
class Bicycle :  public VehicleInterface {
public:
    void drive() override { … }          // Implementierung der abstrakten Methode
};

VehicleInterface *v = new Bicycle();
delete v;                                // ruft Destruktor von Bicycle auf, weil Destruktor
                                         // virtuell ist
```

Automatisch erstellte Methoden

- Vom System zur Verfügung gestellte Konstruktoren, Destruktoren, Zuweisungsoperatoren sind standardmässig nicht virtual!
- Destruktor
  - damit der Destruktor einer abgeleiteten Klasse aufgerufen wird, muss der Destruktor jeder Basisklasse und jedem Interface virtual sein
- Zuweisungsoperator
  - damit bei einer Zuweisung der dynamische Typ berücksichtig wird, muss der Zuweisungsoperator virtual sein
- virtual erzwingen

```
virtual ~Vehicle() = default;        // verwendet Standardimplementierung
virtual Vehicle& operator=(const Vehicle& v) = default;
```

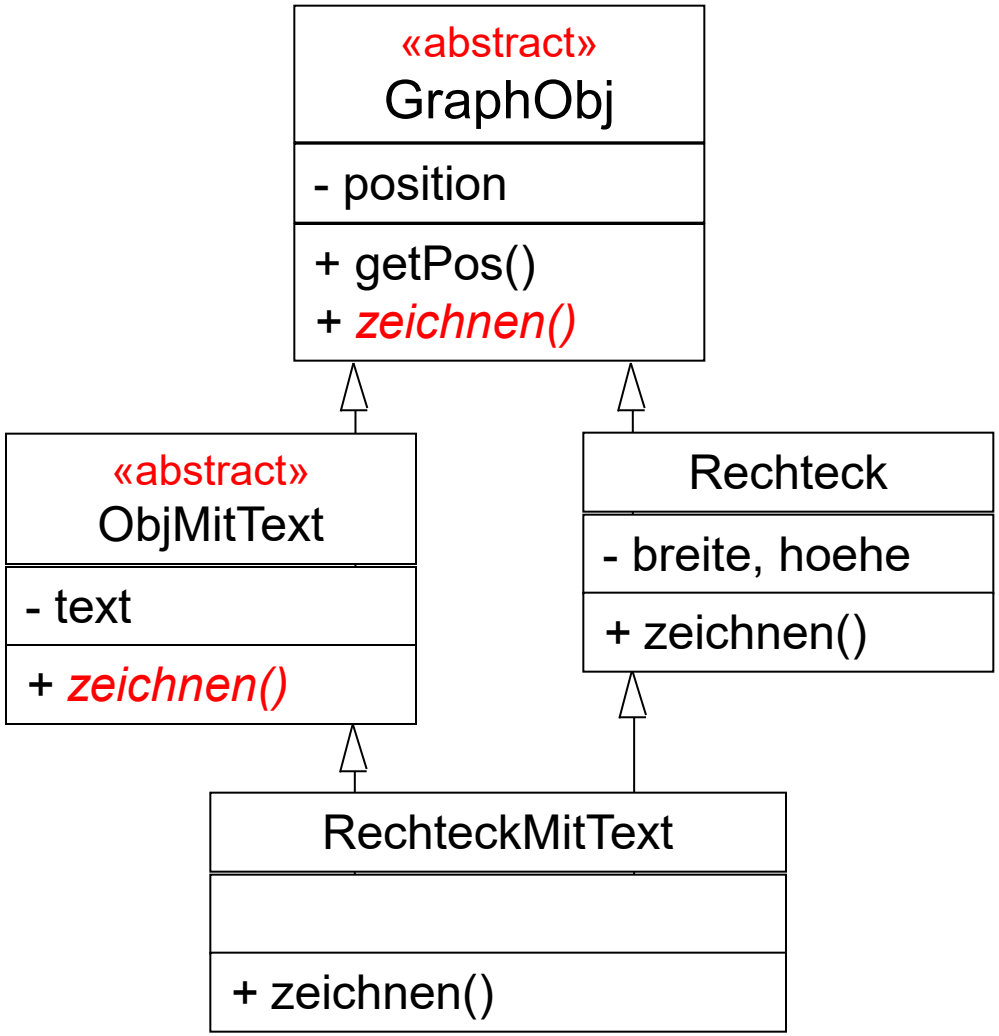
- Best-Practice
  - virtual in Basisklassen/Interfaces für Destruktor und Zuweisungsoperator erzwingen
  - keine virtuellen Methoden in Konstruktoren und Destruktor aufrufen

Destruktoren

- Bei shared_ptr geschieht das Richtige automatisch, d.h. der Basisklassen-Destruktor muss nicht virtuell sein
  - das Ref-Counter-Objekt kennt nur den dynamischen Typ und ruft daher den richtigen Destruktor auf
- Beim Einsatz von unique_ptr sollte der Basisklassendestruktor virtuell sein.
- Achtung!
  - sobald wir virtual ~C() = default; deklarieren, verlieren wir den Verschiebekonstruktor und den Verschiebeoperator. Daher …
- «Rule of Zero» und «Rule of 5 Defaults»
  - wenn nicht nötig, definieren wir keine der fünf Spezialfunktionen (default-ctor, copy-ctor, move-ctor, assignment-op, move-op) und lassen den Compiler diese automatisch generieren
  - wenn wir einen virtuellen Destruktor benötigen, dann definieren wir gleich alle fünf Spezialfunktionen als default, damit wir die Verschiebefunktionen nicht verlieren

Mehrfachvererbung

- Beispiel aus der Welt der grafischen Objekte
- hier mit gemeinsamer Basisklasse
  (ist nicht notwendig)
- Probleme: Namenskonflikte, Mehrdeutigkeiten
- meistens nur für Interfaces sinnvoll



Probleme der Mehrfachvererbung

- Beispiel

```
Rechteck r(0, 0, 20, 50);
RechteckMitText br(10, 5, 60, 60, "Text");
r.zeichnen();  // ruft zeichnen() von Rechteck auf
br.zeichnen() // ruft zeichnen() von RechteckMitText auf

Position rPos = r.getPos();       // gibt Ursprung des Rechtecks zurück
Position brPos = br.getPos();     // → Compiler-Fehler
GraphObj *pObj = &br;             // → Compiler-Fehler
```

- Warum ein Compiler-Fehler?
  - br.getPos() ist nicht eindeutig, denn es könnte getPos() von ObjMitText oder von Reckteck aufgerufen werden
  - Ursache: Teilobjekt GraphObj ist zweimal vorhanden und nicht beide Teilobjekte müssen identisch sein, d.h. die gleiche position besitzen

Lösung: Virtuelle Vererbung

```
class Rechteck : virtual public GraphObj  {
    // Rest normal
};

class ObjMitText : virtual public GraphObj  {
    // Rest normal
};

class RechteckMitText : public ObjMitText, public Rechteck  {
public:
    RechteckMitText(int x, int y, int w, int h, string text)
    : ObjMitText(-2, -2, text), Rechteck(-1, -1, w, h), GraphObj(x, y) {}
    // Rest normal
};
```

Virtuelle Basisklassen und Initialisierung

- Definition
  - vollständiges Objekt: Objekt, das nicht als Teilobjekt dient, also nicht in einem anderen Objekt durch Vererbung enthalten ist
- Ausgangslage
  - virtuelle Basisklassen bewirken, dass nur 1 Teilobjekt dieser Basisklasse in Instanzen einer abgeleiteten Klasse angelegt wird
- Problem
  - welcher Konstruktor ist für die Initialisierung dieses einen Teilobjekts zuständig?
  - im Beispiel: Rechteck(…) oder ObjMitText(…) ?
- Antwort
  - Konstruktor der Basisklasse, welcher im Konstruktor eines vollständigen Objektes aufgerufen wird
  - wird kein Konstruktor der Basisklasse explizit aufgerufen, so wird der Standardkonstruktor der Basisklasse verwendet
  - im Beispiel: GraphObj(x, y) wird verwendet