Technische Hochschule Ingolstadt

Fakultät Informatik

Grundlagen der Programmierung 2

Kapitel 3: Vererbung und abstrakte Klassen

Prof. Dr. Robert Gold | Sommersemester 2025

3.1 Einführendes Beispiel

- Beispiel: *files1.cpp*
- Um das Beispiel zu vereinfachen, wurden die beiden Klassen und das Hauptprogramm in eine Datei geschrieben.
- Die Implementierungen der Methoden stehen direkt in den Klassendeklarationen. Solche Methoden heißen **Inline-Methoden**.
  - Der Compiler ersetzt Aufrufe von Inline-Methoden durch den Methodenrumpf. Bei kurzen Methoden spart das Zeit, weil der Mechanismus zum Funktionsaufruf nicht notwendig ist. Für längere Methoden sollte man das lieber sein lassen.
- Die beiden Klassen ImageFile und TextFile haben den Namen und weitere Attribute und Methoden gemeinsam. Um die Duplizierung von Code zu vermeiden, macht es Sinn, die Gemeinsamkeiten in eine eigene Klasse File herauszuziehen.

3.1 Einführendes Beispiel (Forts.)

- Beispiel: *files2.cpp*
- Die Klasse ImageFile ist von der Klasse File abgeleitet.
- Ein Objekt der Klasse ImageFile hat nun das von der Klasse File geerbte Attribut name und zusätzlich die eigenen drei Attribute. Der Konstruktor von ImageFile hat deswegen wieder vier Parameter.

files2.cpp

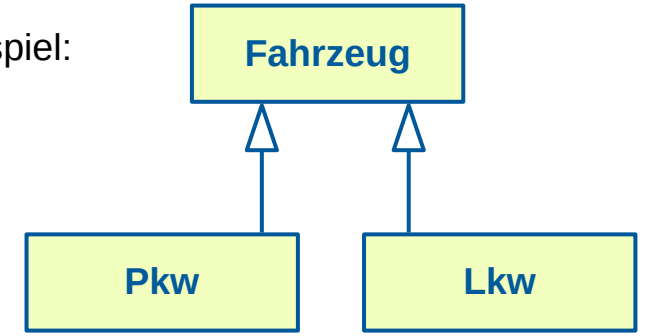
```
class ImageFile : public File
{
private:
    int height;
    int width;
    int bits;
public:
    ImageFile(const string pName, const int pHeight,
                    const int pWidth, const int pBits): File(pName)
    {
        height = pHeight;
        width = pWidth;
        bits = pBits;       // bits per pixel
    }
    int fsize() const { return height * width * bits; }
};
```

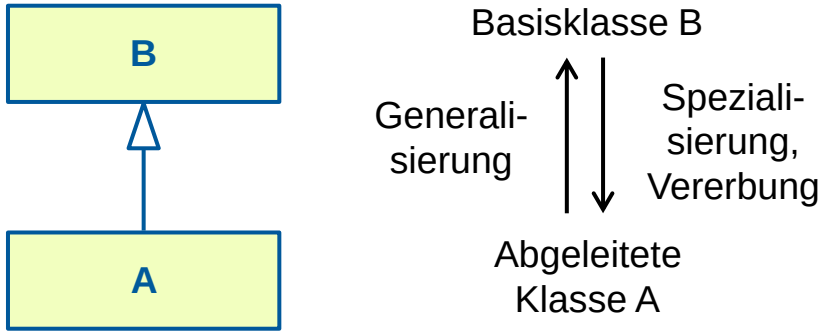
3.2 Vererbung

3.2.1 Definition

- Reale Objekttypen fassen wir im täglichen Leben oft mit Oberbegriffen zusammen. Beispielsweise können wir Pkws, Lkws, Züge unter dem Begriff Fahrzeug einordnen.

Beispiel:



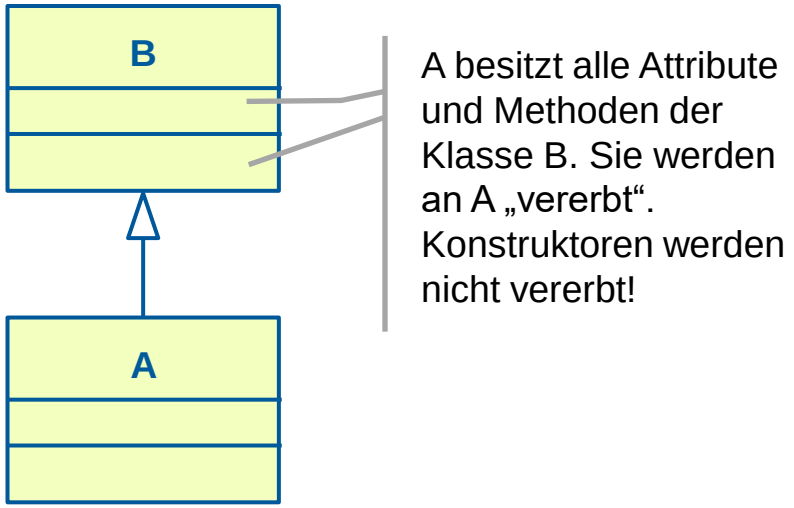


- Wenn Objekte einer Klasse A Spezialfälle von Objekten einer Klasse B sind, ist A eine **Spezialisierung** von B, z.B. ist ein Pkw ein spezielles Fahrzeug.
- Sieht man die Situation aus Sicht von B, heißt B eine **Generalisierung** von A.
- Die Klasse B heißt **Basisklasse**, A wird **abgeleitete Klasse** genannt. Die Bezeichnungsweise ist nicht einheitlich: Statt Basisklasse wird auch Elternklasse oder Superklasse, statt Abgeleitete Klasse wird auch Kindklasse oder Subklasse oder Erweiterung verwendet.

3.2 Vererbung

3.2.1 Definition (Forts.)

- Die abgeleitete Klasse A besitzt damit automatisch alle Attribute und Methoden der Basisklasse B (**Vererbung**), hat aber zusätzliche Attribute und Methoden, die nur für Objekte der Klasse A gelten.
- Eine Spezialisierungs- oder Vererbungsbeziehung bedeutet, dass B durch zusätzliche Attribute und/oder Methoden erweitert wird. Man spricht deshalb auch von einer **Erweiterung** der Basisklasse.

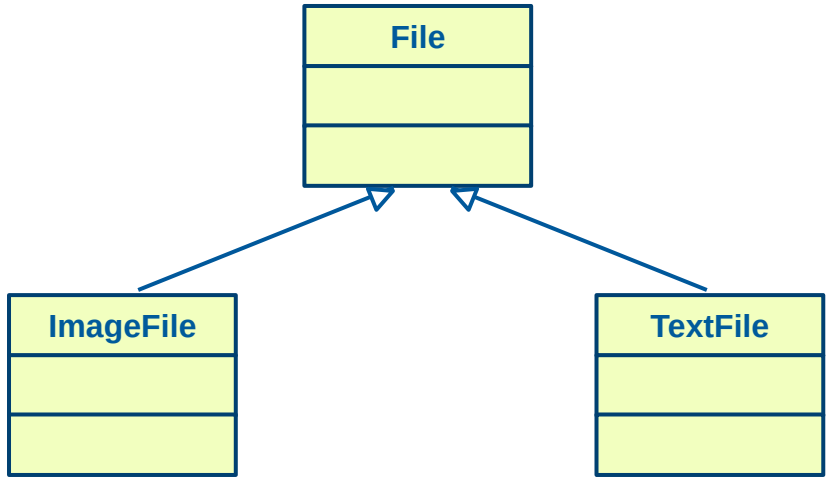


- Im Beispiel erbt die Klasse ImageFile das Attribut name und die Methode getName von der Basisklasse File. Wichtig ist dabei, dass bei der Konstruktion eines Objekts der abgeleiteten Klasse auch die geerbten Attribute initialisiert werden. Im Beispiel:

```
ImageFile(const string pName, const int pHeight,
          const int pWidth, const int pBits): File(pName)
```

Es wird dabei der Konstruktor der Basisklasse aufgerufen.

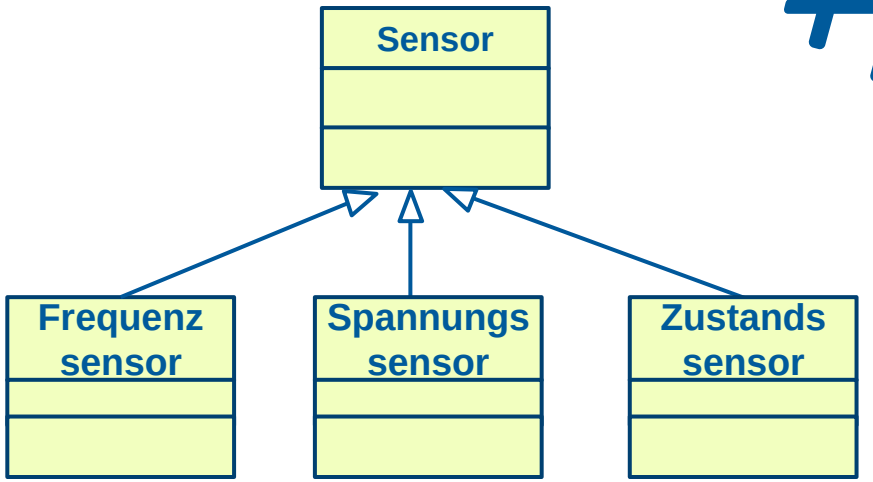
- Vererbung kann eingesetzt werden, wenn eine „**ist-ein**“-Beziehung gilt.
  - Beispiel: Ein ImageFile ist ein File. Deshalb ist ein ImageFile eine Spezialisierung von File. (Oder anders ausgedrückt: ImageFile erbt von File.)
- Vererbung unterstützt die **Wiederverwendung** von Klassen und ist deshalb besonders wichtig.
  - Beispielsweise kann die Klasse File nicht nur für ImageFile sondern auch für TextFile … verwendet werden.
  - Die Wiederverwendung von Klassen reduziert den Entwicklungsaufwand und erhöht die Software-Qualität.

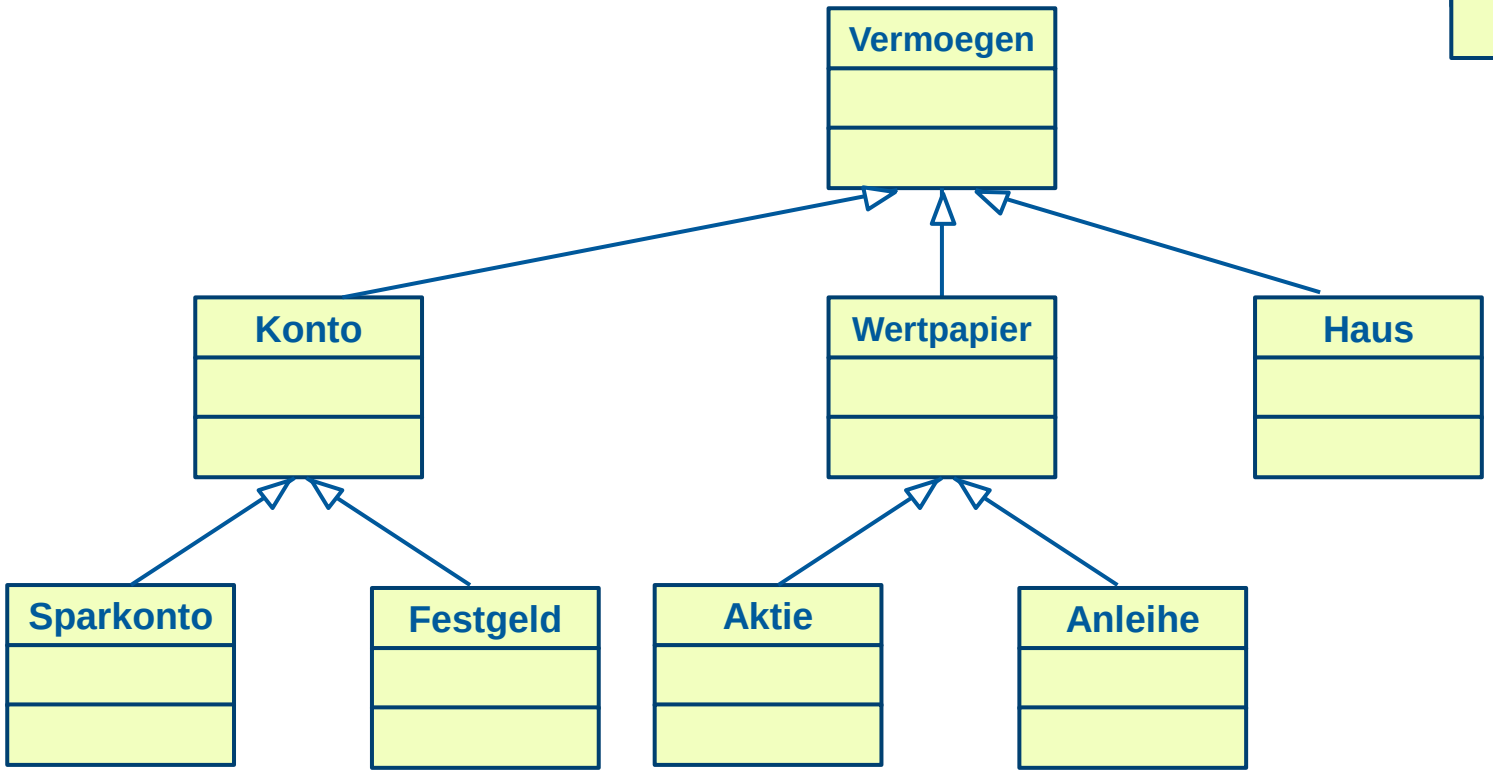


3.2 Vererbung

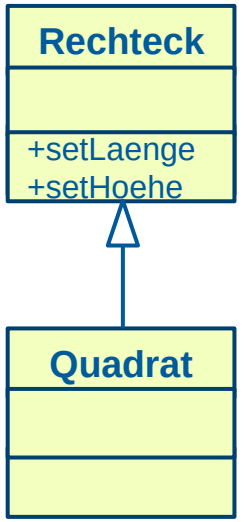
3.2.1 Definition (Forts.)

- Weitere Beispiele:





Schlechtes Beispiel:
Mit Vererbung oder Spezialisierung ist eine Erweiterung der Schnittstelle gemeint. Für ein Quadrat sollte aber setLaenge nicht (ohne gleichzeitige Änderung der Höhe) ausgeführt werden. Dieses Beispiel ist eine Einschränkung und keine Erweiterung.



- Private Attribute und Methoden sind in abgeleiteten Klassen nicht sichtbar. Wenn private Attribute in einer abgeleiteten Klasse verwendet werden sollen, können öffentliche Getter und Setter definiert werden.
- Wenn man das nicht will, können Attribute als **protected** deklariert werden, z.B.

```
class File
{
protected:
    string name;
…
};
```

- Protected Attribute und protected Methoden sind in abgeleiteten Klassen sichtbar und können verwendet werden. Außerhalb der eigenen Klasse und außerhalb abgeleiteter Klassen sind protected Elemente nicht sichtbar.

- In den meisten Fällen werden Klassen öffentlich abgeleitet, z.B.

```
class ImageFile : public File
```

- Es gibt auch die **private Ableitung**, z.B.

```
class ImageFile : private File
```

- Dann sind auch protected Attribute und Methoden in der abgeleiteten Klasse nicht sichtbar.
- Frage: Ein Attribut der Basisklasse soll in abgeleiteten Klassen lesbar aber nicht schreibbar sein. Außerhalb der Basisklasse und außerhalb abgeleiteter Klassen soll das Attribut weder lesbar noch schreibbar sein. Wie kann man das erreichen?

3.3 Virtuelle Methoden und abstrakte Klassen

3.3.1 Überladen und Überschreiben von Methoden

- In der Basisklasse und in abgeleiteten Klassen sind Methoden mit demselben Namen möglich. Wir besprechen zuerst, was geht und was nicht, und danach, wozu das wichtig ist.

B ⟵ A	Variante 1	Variante 2a	Variante 2b	Variante 3	Variante 4	Variante 5
in B		int f(int x)	int f(int x)	int f(int x)	int f(int x)	virtual int f(int x)
in A	int f() int f(int x)	int f(int x)	int f(int x)	int f()	int f(int x)	virtual int f(int x)

Aufruf	A a;
	a.f();
	a.f(2);
		A a;
		a.f(2);
			A a;
			a.B::f(2);
				A a;
				a.f(2);
					A a;
					B b;
					B *p = &b;
					p->f(2);
					p = &a;
					p->f(2);
						A a;
						B b;
						B *p = &b;
						p->f(2);
						p = &a;
						p->f(2);

aufgerufen wird	f() in A f(2) in A	f(2) in A	f(2) in B	Error	f(2) in B f(2) in B	f(2) in B f(2) in A
	Überladen	Überschreiben (f in B verdeckt)	Überschreiben (f in B verdeckt)			Polymorphie

- **Überladen**: Zwei Funktionen haben denselben Funktionsnamen, aber unterschiedliche Parametersignatur.
  - Verwendung: Wenn zwei Operationen dieselbe Bedeutung haben, aber unterschiedliche Parametersignaturen verwenden, müssen nicht verschiedene Namen gewählt werden.
- **Überschreiben**: Eine Funktion in einer abgeleiteten Klasse hat denselben Funktionsnamen wie eine Funktion der Basisklasse.
  - Verwendung: Die Operation wird in der abgeleiteten Klasse angepasst und ist unter demselben Namen verfügbar.
- **Polymorphie**: Wie Überschreiben aber mit identischer Parametersignatur und Rückgabetyp und dem Schlüsselwort virtual (**virtuelle Methode**), aber es wird erst zur Laufzeit anhand des Objekttypen ausgewählt, welche Funktion aufgerufen wird.
  - Verwendung: Z.B. beim Durchlaufen einer Liste von Objekten von abgeleiteten Klassen (siehe unten)

- Beispiel: *files3.cpp*

files3.cpp

```
class File
{
    …
    virtual int fsize() const { return 0; }
};

class ImageFile : public File
    …
    virtual int fsize() const override
        { return height * width * bits; }
};
```

files3.cpp

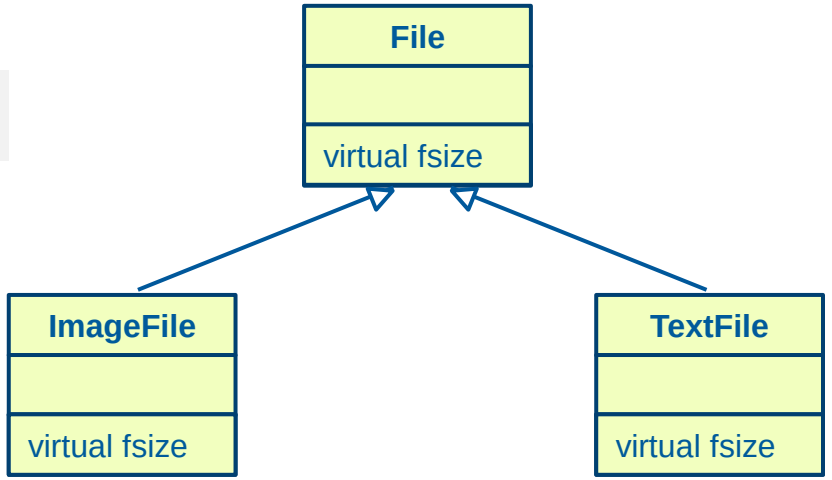
```
class TextFile : public File
{
    …
    virtual int fsize() const override
        { return chars * bytes * 8; }
};

int main()
{
    ImageFile *ifile = new ImageFile("image.bmp", 480, 640, 16);
    TextFile *tfile = new TextFile("text.txt", 1000, 2);
    vector<File*> files = { ifile, tfile };
    for (auto fp: files)
        cout << fp->getName() << " " << fp->fsize() << '\n';
}
```

In die Liste files können beliebige Dateien eingefügt werden. Durch fp->fsize() wird zur Laufzeit die richtige Funktion aufgerufen.

- Im Beispiel könnte sogar nachträglich noch eine weitere File-Klasse eingefügt werden, ohne dass die Schleife geändert werden muss. Die Definition

```
virtual int fsize() const { return 0; }
```

in der Basisklasse gibt somit die erforderliche **Schnittstelle** vor, die alle abgeleiteten Klassen implementieren müssen.



- override in den abgeleiteten Klassen ist nicht erforderlich, aber empfehlenswert, weil der Compiler prüfen kann, ob es in der Basisklasse eine gleichlautende Funktion gibt.
- virtual in den abgeleiteten Klassen ist ebenfalls nicht erforderlich, aber auch empfehlenswert, weil dadurch klar wird, dass es sich um eine virtuelle Methode handelt.
- Polymorphie funktioniert nur mit Zeigern oder Referenzen und dem Schlüsselwort virtual. Sonst ist der Objekttyp festgelegt.

- Realisierung von Polymorphie in C++:
  - „[…] indem vom Compiler im Speicherbereich eines Objekts zusätzlich zu den Objektattributen ein Zeiger vptr auf eine besondere Tabelle vtbl (virtual table = Tabelle von Zeigern auf virtuelle Funktionen) eingebaut wird.
  - Die Tabelle gehört zu der Klasse des Objekts und enthält ihrerseits Zeiger auf die virtuellen Funktionen dieser Klasse.
  - Wenn nun eine virtuelle Funktion über einen Zeiger oder eine Referenz auf dieses Objekt angesprochen wird […,] wird damit die zu diesem Objekt gehörende Funktion aufgerufen.
  - Wenn die Klasse dieses Objekts aber keine Funktion mit gleicher Signatur hat, wird die entsprechende Funktion der Oberklasse gesucht und aufgerufen.
  - […] Objekte [werden] durch den versteckten Zeiger vptr etwas größer [,…] der Zugriff auf virtuelle Funktionen [dauert] durch den Umweg über die Zeiger geringfügig länger […].“ [1, S. 316]

- Vererbung bedeutet eine „ist-ein“-Beziehung, z.B. ein TextFile ist ein File. Die abgeleitete Klasse ist ein Subtyp der Basisklasse. Deshalb kann man dort, wo ein Objekt der Basisklasse erwartet wird, auch ein Objekt der abgeleiteten Klasse einsetzen (**Liskovsches Substitutionsprinzip**).
- Beispielsweise kann eine Funktion mit einem Parameter vom Typ der Basisklasse auch mit einem Objekt der abgeleiteten Klasse aufgerufen werden. Beispiel:

```
void printName(const File file)
{
    cout << "File name: " << file.getName() << '\n';
}

// Im Hauptprogramm
TextFile tfile("text.txt", 1000, 2);
printName(tfile);
```

- Das Liskovsche Substitutionsprinzip gilt auch bei Zuweisungen. Beispiel:

```
TextFile tfile("text.txt", 1000, 2);
File file("noname");
file = tfile;
```

Aber Vorsicht! Durch die Zuweisung file = tfile; wird nur der Basisklassenanteil kopiert. Die Attribute chars und bytes sind für file nicht verfügbar (auch nicht, wenn Getter dafür in TextFile definiert werden).

- Hier helfen Referenzen (oder Pointer) und Polymorphie. Beispiel:

```
TextFile tfile("text.txt", 1000, 2);
File& file = tfile;
cout << file.fsize() << '\n';    // Es wird die Funktion fsize in TextFile ausgeführt
```

- Wenn ein Objekt der abgeleiteten Klasse mit new erzeugt wird und einem Basisklassenzeiger zugewiesen wird, wird bei delete der Destruktor der Basisklasse aufgerufen, z.B. [1, S. 325].

```
Basis* pba = new Abgeleitet(3.0, 3.3);
delete pba; // ok nur mit virtuellem Destruktor!
```

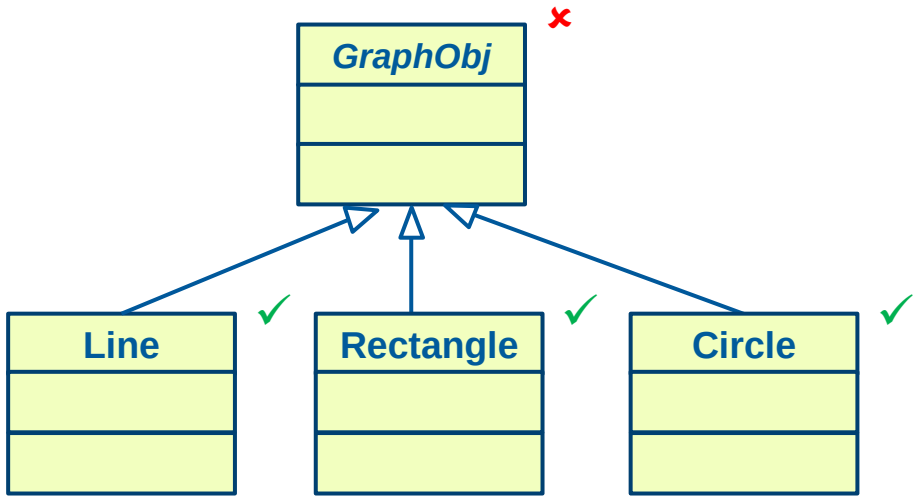
Deshalb muss der Destruktor virtuell sein, damit Polymorphie greift!

```
virtual ~Basis()
{   }
```

3.3 Virtuelle Methoden und abstrakte Klassen

3.3.3 Abstrakte Klassen

- Oft soll von einer Basisklasse gar kein Objekt erstellt werden. Die Basisklasse ist nur zum Ableiten da. Eine solche Klasse heißt **abstrakte Klasse**.
- Beispiel: Ein grafisches Objekt, das keine spezielle Form (Linie, Rechteck, Kreis) hat, gibt es gar nicht. Deshalb sollte kein Objekt von dieser Klasse erstellt werden.
- In C++ ist eine Klasse abstrakt, wenn sie eine **rein virtuelle Methode** enthält, z.B.




```
virtual int fsize() const = 0;
```

Eine rein virtuelle Methode wird mit = 0 gekennzeichnet.
Sie kann eine Definition in der Basisklasse haben, muss aber nicht.

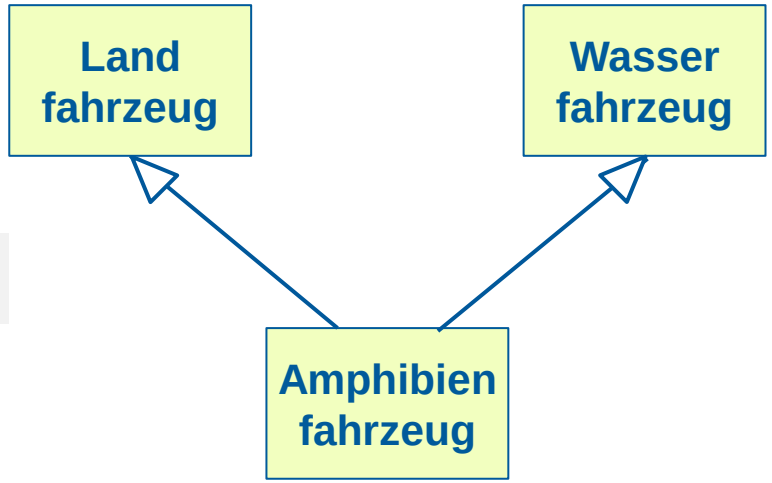
- Beispiel: *files4.cpp*
- In Klassendiagrammen werden abstrakte Klassen durch Kursivdruck gekennzeichnet.
- Vorsicht:
  - Wenn in der abgeleiteten Klasse eine Implementierung einer rein virtuellen Methode fehlt, ist die abgeleitete Klasse ebenfalls abstrakt und es können keine Objekte erstellt werden.
  - Es kann keine Funktion geben, die ein Objekt einer abstrakten Klasse als Werteparameter hat. Als Referenzparameter ist das erlaubt.

3.4 Mehrfachvererbung

- Eine Klasse kann auch mehrere Basisklassen haben (**Mehrfachvererbung**).
- Beispiel:

  Ein Amphibienfahrzeug ist ein Landfahrzeug und ein Wasserfahrzeug.

```
class Amphibienfahrzeug : public Landfahrzeug, public Wasserfahrzeug
```



- Mehrfachvererbung wird selten benötigt und bringt Probleme mit sich. Deshalb gehen wir nicht näher darauf ein. Andere Programmiersprachen, z.B. Java, verbieten die Mehrfachvererbung. Weitere Details siehe [1, S. 333-339]

3.5 Typinformation zur Laufzeit

- Insbesondere für Tests ist es praktisch, den Typ eines Objekts zur Laufzeit abzufragen. Dazu gibt es die Funktion **typeid**. Erforderlich ist #include <typeinfo>.
- Beispiel:

```
ImageFile ifile("image.bmp", 480, 640, 16);
TextFile tfile("text.txt", 1000, 2);
vector<File*> files = { &ifile, &tfile };

cout << (typeid(tfile) == typeid(TextFile)) << '\n';      // Der Typ des Objekts tfile wird mit dem Typ
                                                          // der Klasse verglichen. Ausgabe: 1
for (File *fp: files)
    cout << (typeid(*fp) == typeid(TextFile)) << '\n';    // Hier kommt Polymorphie zum Tragen. Ohne Polymorphie
                                                          // wäre *fp vom Typ File. Ausgabe: 0 1
```