Objektorientierung: Klassen und Objekte

- Klasse:
  - Beschreibung für eine Menge von Objekten
  - Schablone, Bauplan
  - abstrakte Form
- Objekt:
  - Instanz einer Klasse
  - konkreter Inhalt (Werte)

Klassen

- bestehen aus
  - Attributen /Eigenschaften
  - Methoden (Funktionen)
- Aufteilung in
  - öffentliche / nach außen sichtbare Teile, d.h. Zugriff von Objekten anderer Klassen möglich (**public**)
  - private / nach außen verborgene Teile, d.h. Zugriff nur von Objekten der eigenen Klasse (**private**)
- automatische Initialisierung von Attributen
  - Zahlen z.B. werden auf 0 gesetzt
  - besser selbst für Initialisierung sorgen

Klassen: Beispiel

```
class Person
{
    private:
        unsigned int age;  // Attribut Alter ist privat

    public:
        std::string name;   // Attribut Name ist öffentlich

        // Deklaration von Methoden
        int getAge();           // Methode holeAlter ist öff.
        void setAge();          // Methode setzeAlter ist öff.
}; // !!! Semikolon beendet Klassenbeschreibung
```

Beispiel: erweitert

```
class Person
{
    private:
        ...

        // Definition der Methoden
        int getAge() {return age;}
        void setAge(unsigned int a) {age = a;}
};
```

Header / Source -Datei

- Header-Datei “XXX.h”
  - enthält Struktur
  - Deklaration von Attributen der Klasse
  - Deklaration von Methoden der Klasse
- Quell-Datei “XXX.cpp”
  - füllt Struktur
  - füllt Methoden mit Inhalt
  - definiert Initialwerte für bestimmte Attribute
- XXX entspricht üblicherweise dem Namen einer Klasse, die in der Datei definiert wird

Beispiel Header Datei

Person.h

```
class Person
{
    private:
        unsigned int age;

    public:
        std::string name;

       // Deklaration von Methoden
        int getAge();
        void setAge();
};
```

Beispiel Source Datei

Person.cpp

Person::getAge()

:: -Operator
hat Priorität 1

```
#include “Person.h”

// Definition der Methoden

int Person::getAge()
{
    return age;
}

void Person::setAge(unsigned int a)
{
    age = a;
};
```

#include

- #include <header1.h>
  - wird in den durch INCLUDEPATH definierten Verzeichnissen gesucht
- #include “header2.h”
  - wird auch im aktuellen Verzeichnis gesucht

g++ Option (`-I`)

- `-I` ***`INCLUDEPATH`*** gibt dem Präprozessor an wo er nach Header Dateien suchen soll.

#define Direktive

- definiert ein Macro für den Präprocessor
- wenn der Präprocessor anschliessend auf den Macro Namen trifft wird dieser durch seinen Wert ersetzt

```
#define TEST
#define TEST 1
#define NAME          test
#define SQUARE(X)   X*X
```

g++ Option -D

- erlaubt es ein Makro beim Aufruf des g++ zu deklarieren
- -D hat als zusätzliches Argument den Namen des Makros und optional einen Wert

  g++ -DMEIN_MAKRO=3 ....

#ifdef...#endif Direktiven

- mit der **`#ifdef...#endif`** Direktive lassen sich Codezeilen in Abhängigkeit eines Makros von der Prozessierung durch den Präprozessor ausschließen

```
#ifdef AUSGABE_GEWUENSCHT
#include <iostream>
std::cout << "Ausgabe\n";
#endif
```

#ifndef...#endif Direktiven

- mit der **`#ifndef...#endif`** Direktive lassen sich Codezeilen in Abhängigkeit eines Makros von der Prozessierung durch den Präprozessor ausschließen

```
#ifndef KEINE_AUSGABE_GEWUENSCHT
#include <iostream>
std::cout << "Ausgabe\n";
#endif
```

#else Direktive

- entsprechend der **`if`** Anweisung gibt es auch für **`#ifdef`** und **`#ifndef`** eine **`#else`** Direktive

```
#ifdef TEST
...
#else
...
#endif
```

#ifndef in Header Dateien

- Die Deklarationen in fast allen Header Dateien sind in **`#ifndef...#endif`** Direktiven eingeschlossen
- Direkt nach der **`#ifndef`** folgt eine **`#define`** Direktive, die das Makro, welches von der **`#ifndef`** Direktive davor geprüft wird definiert.

Header Struktur

```
// Header für Klasse Rechteck
#ifndef RECHTECK_H__
#define RECHTECK_H__

DEKLARATIONEN

#endif // RECHTECK_H__
```

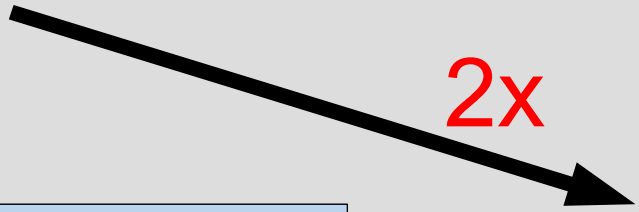
#include Problem 1/2

form.h

```
class form
{
   ...
};
```

kreis.h

```
#include “form.h”
class kreis : public form
{
   ...
};
```



main.cpp

```
#include “form.h”
#include “kreis.h”

int main()
{
    form f;
    kreis k;
}
```

Expandierter Sourcecode

#include Problem 2/2

form.h

```
#ifndef FORM_H__
#define FORM_H__
class form
{
   ...
};
#endif
```

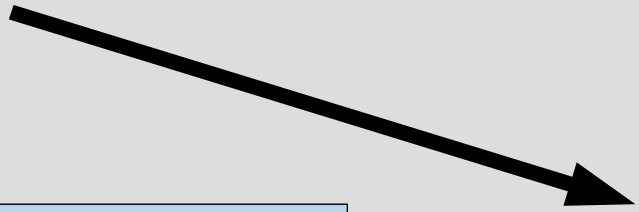
kreis.h

```
#include “form.h”
class kreis : public form
{
   ...
};
```

main.cpp

```
#include “form.h”
#include “kreis.h”

int main()
{
    form f;
    kreis k;
}
```



Expandierter Sourcecode

Deklaration und Zugriff

```
Person  p1, p2;   // p1 und p2 sind Instanzen der
                  // Klasse Person, Speicher wird
                  // alloziert und zugewiesen

// Punktoperator für den Zugriff  auf Attribute und
// Methoden der Klasse
p1.name = “Tom”;
p2.name = “Alex”;

std::cout   <<  p1.name << “\n”;   // Attribut ist public
std::cout   <<  p1.getAge()   << "\n";
```

Konstruktor

- dient dazu, Objekt in definierten Anfangszustand zu versetzen
  - Speicherplatz für die Attribute bereitstellen
  - ggf. Initialisierung der Attribute mit geeigneten Werten
- Konstruktoren haben den gleichen Namen wie die Klasse
- Konstruktoren haben keinen Rückgabewert

Beispiel Klasse Datum

```
class Datum
{
private:
    unsigned int t,m,j;

public:
    void setze (unsigned int _t, unsigned int _m, unsigned int _j);
    void setzeAufHeute();
};
```

Datumsobjekt sollte stets initialisiert werden können.

Konstruktoren: Deklaration

- Standard Konstruktor / default constructor

```
public:
    Datum();
```

Besonderheit: wird automatisch implizit definiert, wenn kein Konstruktor angegeben wird

- Allgemeine Konstruktoren

```
public:
    Datum(unsigned int _t, unsigned int _m, unsigned int _j);
    Datum (unsigned int _j);
```

Deklaration in der Header Datei

Konstruktoren: Definition

```
Datum::Datum()
{
    setze(1, 1, 2000);
}

Datum::Datum(unsigned int _t, unsigned int _m, unsigned int _j);
{
    setze(_t, _m, _j);
}
Datum (unsigned int _j)
{
    setze (1, 1, _j);
}
```

Definition in der Quell Datei

Ablauf beim Aufruf eines Konstruktors

- Konstruktor der Basisklasse wird aufgerufen
- Aufruf der Konstruktoren aller Attribute, die selbst Objekte sind (in der Reihenfolge der Deklaration) sowie Standard-Initialisierung der Attribute, die sich auf primitive Datentypen beziehen
- Abarbeitung des Methodenkörpers des Konstruktors der abgeleiteten Klasse

Initialisierungsliste

- Initialisierung von Attributen bei der Erzeugung eine Objekts

statt Erzeugung des Objekts + nachfolgende Initialisierung

```
Datum::Datum(unsigned int _t, unsigned int _m, unsigned int _j);
{
    setze(_t, _m, _j);
}
```

Initialisierung über Liste

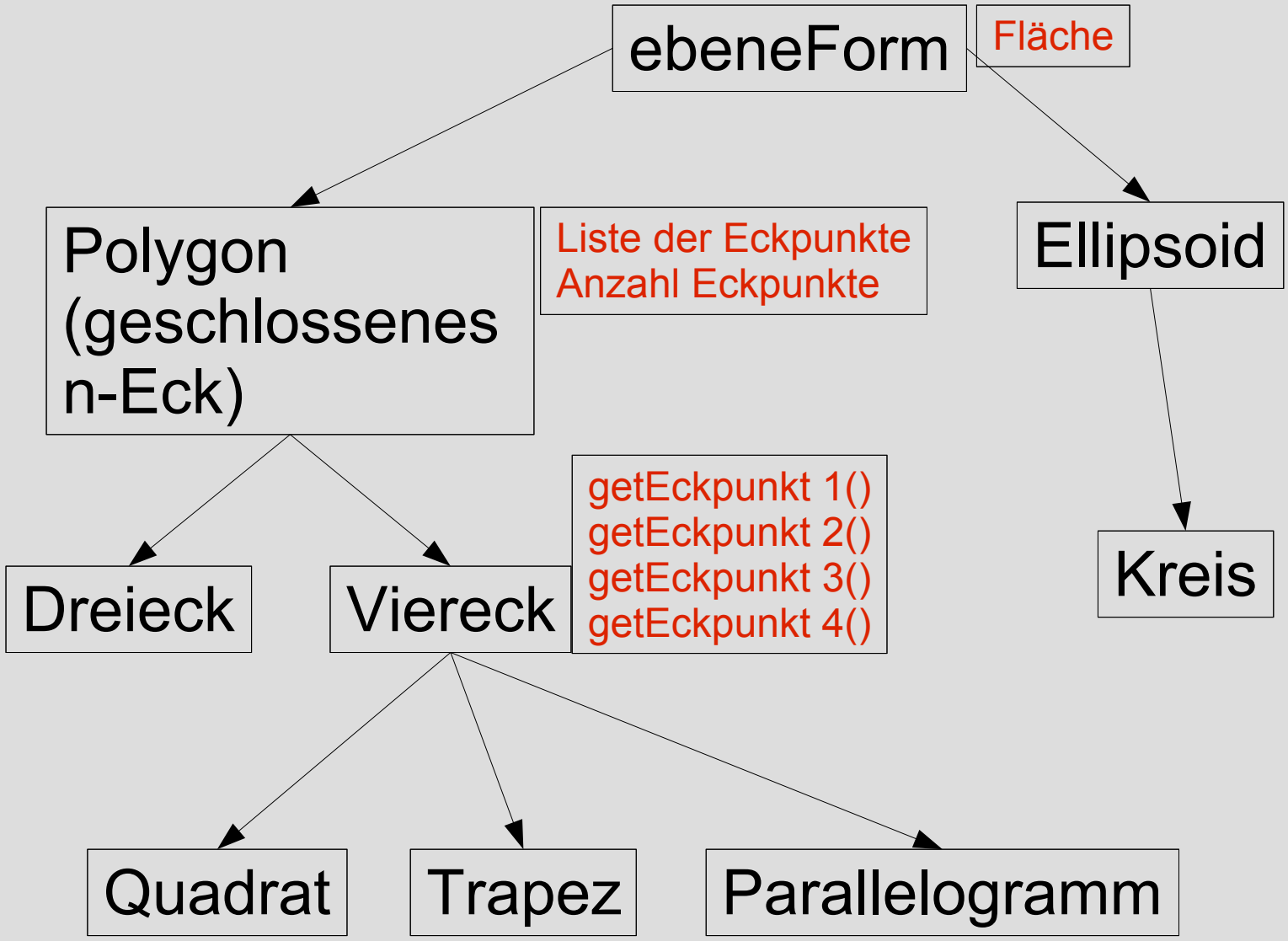
```
Datum::Datum(unsigned int _t, unsigned int _m, unsigned int _j)
: t(_t), m(_m), j(_j);
{
}
```

Vererbung

Definition von Eigenschaften

ererbte Eigenschaften

Beispiel:Klassenhierarchie



Vererbung

- Wiederverwendung von Elementen einer Klasse durch Weitergabe an Unterklassen -> Vererbungshierarchie
- Vorteile:
  - Zusammenfassung von Gemeinsamkeiten, die so nur an einer Stelle im Programm vorhanden sind
  - bei Änderungen muß nur an einer Stelle modifiziert werden
  - Funktionalität kann immer wieder verwendet werden
  - Hinzufügen von Funktionalität wird erleichtert (Aufbau auf Basisfunktionalität)

Deklaration einer abgeleiteten Klasse

```
class Viereck : public ebeneForm
{

};
```

```
class Quadrat : public Viereck
{

};
```

Beispiel: Vererbung

A.h

```
#ifndef A_H
#define A_H

class A
{
    public:
        A();
        int a;
};

#endif
```

A.cpp

```
#include “A.h”

A::A()
{
    a = 0;
}
```

B.h

```
#ifndef B_H
#define B_H

class B : public A
{
    public:
        B();
};

#endif
```

B.cpp

```
#include “B.h”

B::B()
{
    a = 1;
}
```

Beispiel: Vererbung

Main.cpp

```
#include “A.h”
#include “B.h”

int main ()
{
    A ele1
    B ele2;
    std::cout << “a: “ << ele1.a << “\n”;
    std::cout << “a: “ << ele2.a << “\n”;
}
```

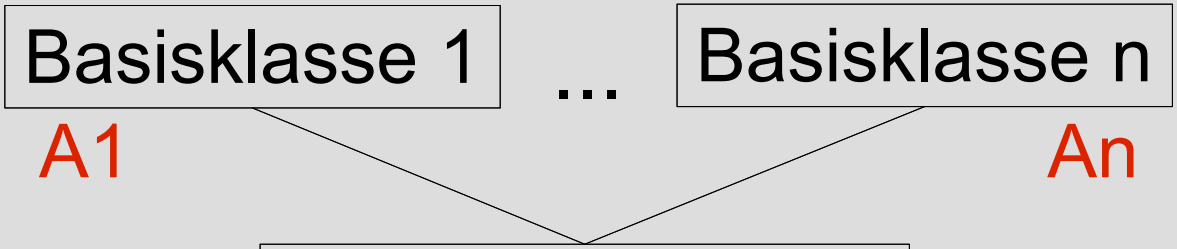
Übersetzung: g++ A.cpp B.cpp Main.cpp -o TestVererbung
Ausführung: ./TestVererbung

Zugriffsbeschränkungen

- **public**: Elemente und Methoden unterliegen keiner Zugriffsbeschränkung
- **private**: Elemente und Methoden sind nur innerhalb der Klasse selbst zugreifbar
- **protected**:Elemente und Methoden sind ausschließlich innerhalb der Klasse selbst und allen mit public davon abgeleiteten Klassen zugreifbar

Mehrfachvererbung

- eine Klasse kann von mehreren anderen Klassen abgeleitet sein
- sie erbt dann die Elemente aller Basisklassen




```
class B : public A1, ..., public An
{
    ..
};
```

Mehrfachvererbung

- problematisch wegen
  - Tendenz zu Unübersichtlichkeit
  - Namensgleichheit muß explizit vermieden werden
- sollte vermieden werden
- in anderen Sprachen wie z.B. in Java, ist Mehrfachvererbung gar nicht erst erlaubt

Überschreiben von Methoden

- jede Klasse erbt die Methoden ihrer Basisklasse
- zusätzliche Eigenschaften der abgeleiteten Klasse (Unterklasse) führen dazu, daß es u.U. zusätzliche Parameter gibt, die die Methode beeinflussen => Methode muß in der Unterklasse modifiziert werden, d.h. übrschrieben. Zwei Möglichkeiten
  - Methode wird komplett ersetzt
  - Methode in der abgeleiteten Klasse ruft Methode der Basisklasse auf und führt dann weitere Befehle aus

Eigenschaften OO

- Kapselung / Information Hiding
  - Klassen-Interna sind nach außen unsichtbar und können daher leicht verändert werden
- Vererbung
  - Eigenschaften der Basisklasse stehen automatisch in abgeleiteten Klassen zur Verfügung
- Polymorphismus