Hochschule Esslingen
University of Applied Sciences

Objektorientierte Programmierung mit C++:

Besondere Klassenelemente

Profs. M. Dausmann
D. Schoop, A. Rößler
M.Sc. A. Freymann

Fakultät Informationstechnik, Hochschule Esslingen

Agenda

- Statische Klassenelemente (Datenelemente, Funktionen)
- Konstante Datenelemente in Objekten
- Konstante Objekte
- Konstante Instanzfunktionen
- Freundfunktionen, Freundklassen
- Objekte als Datenelemente einer Klasse
- Aggregation und Komposition

Statische Klassenelemente

- Ausgangssituation:
  - Instanz = Objekt
  - Instanzmethode = Objektmethode
  - Klassen definieren Instanzvariablen und Instanzmethoden
  - Jedes Objekt einer Klasse besitzt eigene Instanzvariablen und nutzt die Instanzmethoden der Klasse. D.h. zwei Objekte einer Klasse haben Variablen mit den gleichen Namen, die aber unterschiedliche Speicherbereiche belegen.
- Manchmal möchte man Daten über die Menge aller Objekte einer Klasse speichern. Dazu dienen dann:

  → statische Datenelemente, statische Methoden

Beispiel Tagegeldkonto (1)

- Folgende Vorstellung: Eine Klasse **TagegeldKonto** enthält den Kontostand und den aktuellen Zinssatz. Jede Instanz entspricht einem Sparkonto.
- Ändert sich der Zinssatz, muss die Änderung mühsam in jeder Instanz durchgeführt werden.
- Der Einsatz einer globalen Variablen kommt wegen des Prinzips der Datenkapselung nicht in Frage.
- Die Lösung ist eine Variable, die pro Klasse nur einmal vorhanden ist. Diese Variable ist in jeder Instanz sichtbar, kann aber nach außen gekapselt werden (private!).
- Diese Art von Variable nennt sich **static**-Attribut/Variable oder auch Klassenattribut/Klassenvariable.

Beispiel Tagegeldkonto (2)

```
class TagegeldKonto
{
      char name[50];
      float total;
      static float zinssatz; //Klassenattribut
public:
      TagegeldKonto();
      void addiereZins(){
            total = total * (1 + zinssatz);
      }
};
```

Klassenvariablen

- Statische Datenelemente (= Klassenvariablen) beschreiben Eigenschaften einer Klasse (die Menge aller Objekte) und nicht Eigenschaften von einzelnen Objekten.
- Sie verhalten sich wie globale Variablen, auf die jedes Klassenobjekt zugreifen kann.
- Sie werden in der Klasse deklariert und außerhalb initialisiert:

```
float TagegeldKonto::zinssatz = 0.0075;
```

- Sie können benutzt werden, auch wenn es noch kein Objekt der Klasse gibt.

Beispiel für Klassenvariablen

```
class Schueler {
  int schuelerNr;
  static int schuelerAnz;
public:
  Schueler() {
    schuelerNr = ++schuelerAnz;       // schuelerAnz wird bei
                                      // jedem neuen Schüler erhöht
  }
  void abzaehlen() {
    cout << "Ich bin die Nummer " << schuelerNr << endl;
  }
};
// Initialisierung ausserhalb der Klasse!
int Schueler::schuelerAnz = 0;
void main() {
  Schueler klasse [10]; // 10 Objekte der Klasse Schueler instantiiert
  for (int i = 0; i < 10; i++) {
    klasse[i].abzaehlen(); }
}
```

Beispiel: statische Methoden (1)

```
class Schueler {
  int schuelerNr;
  static int schuelerAnz;
public:
  Schueler() {
    schuelerNr = ++schuelerAnz;
  }
  void abzaehlen() {        // Instanzmethode
    cout << "Ich bin die Nummer " << schuelerNr << endl;
  }
  static void ergebnis() { // statische Methode
    cout << "Schuelerzahl: " << schuelerAnz << endl;
  }
};                          // statische Methode
```

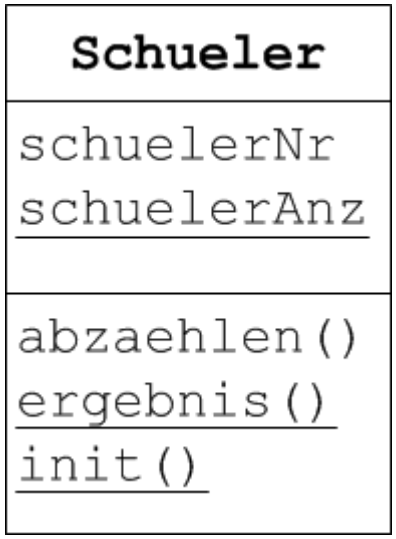
Beispiel: statische Methoden (2)

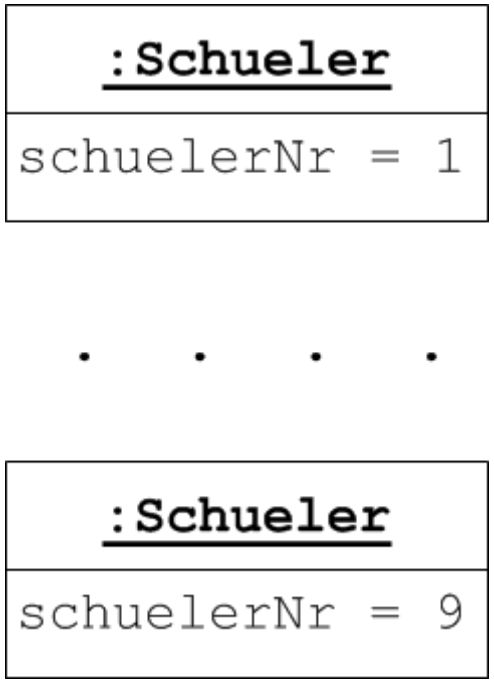
```
int Schueler::schuelerAnz = 0; // Deklaration notwendig!!!

void main() {
  Schueler::ergebnis();
  Schueler klasse [10];     // jetzt Objekte vorhanden
  // schueler[1].init();    // Init auch hier möglich
  for (int i = 0; i < 10; i++)
    klasse[i].abzaehlen(); // Instanzmethode
  for (i = 0; i < 10; i++)
    klasse[i].ergebnis();  // statische Methode
}
```

Statische Funktionen

- Statische Methoden dürfen nur auf statische Daten-elemente zugreifen.
- Sie können daher ohne Objekt aufgerufen werden. Sie verfügen über keinen this-Zeiger.





Statische Methoden

- Statische Methoden dürfen nur auf statische Daten-elemente zugreifen.
- Sie verfügen über keinen this-Zeiger.
- Sie können aufgerufen werden, auch wenn es (noch) kein Objekt der Klasse gibt.
- Beim Aufruf einer statischen Methode stellt man den Klassennamen mit dem Scope-Operator vor den Namen der Methode.

Zugriff auf Datenelemente

- Auf statische Datenelemente können zugreifen:
  - statische Methoden
  - Instanzmethoden
  - Freundfunktionen (kommen später dran)
- Auf nichtstatische Datenelemente können nur nicht-statische ("normale") Methoden zugreifen (this-Zeiger).
- Eine statische Methode oder eine Freundfunktion kann auf nichtstatische Datenelemente eines Objektes der Klasse zugreifen, das man ihr als Parameter übergeben hat.

Beispiel: Zugriff auf nichtstatische Datenelemente

```
class Schueler {
  int schuelerNr;
  static int schuelerAnz;
public:
  Schueler() { ... }
  void abzaehlen() const { ... }
  static void ergebnis() { ...}
  static void print(const Schueler & s) {// statische Methode, der
    cout << schuelerAnz << endl;          // Objekt übergeben wird.
    cout << s.schuelerNr << endl;         // Zugriff auf Instanz-
  }                                       // variable wird möglich.
};
int Schueler::schuelerAnz = 0;
void main() {
  Schueler klasse [10];
  Schueler::print(klasse[5]);
}
```

Statische vs. nichtstatische Methoden

- Statische Methoden werden immer statisch gebunden. Es gibt damit beim Aufruf keinen Overhead für dynamisches Binden (virtuelle Methoden, Polymorphie (kommt später)).
- Beim Aufruf einer statischen Methode entfällt die Verwaltung des this-Zeigers.
- Will man zusammengehörende globale Funktionen zusammenfassen, kann man sie als statische Methoden einer Klasse realisieren:

```
class Math { public:
    static double sin(double x);
    static double cos(double x);
    ……
```

Initialisierungsliste

- Eine Initialisierung ist *keine* Zuweisung. Jede Zuweisung muss sich auf ein bereits existierendes Objekt beziehen.
- Der Ablauf der Initialisierung geschieht folgendermaßen:
  1. Zuerst werden noch vor Betreten des Anweisungsblocks die Datenelemente angelegt. Dabei wird Speicher angefordert und deshalb der zugehörige Konstruktor aufgerufen. Anschließend wird die Initialisierungsliste in der Reihenfolge abgearbeitet, in der die Elemente in der Klassendeklaration aufgeführt sind.
  2. Danach wird der Anweisungsblock ausgeführt.
- Verwendet man also die Initialisierungsliste, dann erfolgt die Initialisierung bereits beim Anlegen des Objektes.
- Dies bringt zum Teil erhebliche **Effizienzvorteile** mit sich.

Beispiel: Dreieck aus Punkten (1)

```
class Punkt {
  double x, y;
public:
   Punkt(double x = 0, double y = 0) : x(x), y(y) { .. }

  ~Punkt(){ .. }
  double getX() { return x };
  double getY() { return y };
  void setXY(double x, double y) { ..};
}
```

Beispiel: Dreieck aus Punkten (2) mit Initialisierungsliste

```
class Dreieck {
  Punkt p1; Punkt p2; Punkt p3;
public:
  Dreieck(double x1, double y1, double x2,
          double y2, double x3, double y3);
...};

Dreieck::Dreieck(double x1, double y1, double x2,
                 double y2, double x3, double y3):
                 p1(x1,y1), p2(x2,y2), p3(x3,y3){};
```

Aufruf:

```
Dreieck d(-3,0,0,4,3,0);
```

Ausgabe:

```
Konstruktor von (-3, 0)
Konstruktor von (0, 4)
Konstruktor von (3, 0)
```

Beispiel: Dreieck aus Punkten (3) ohne Initialisierungsliste

```
class Dreieck {
  Punkt p1; Punkt p2; Punkt p3;
public:
   Dreieck(double x1, double y1, double x2,
           double y2, double x3, double y3) {
           p1.setXY(x1,y1);
           p2.setXY(x2,y2);
           p3.setXY(x3,y3); }
   ...
};
```

Aufruf:

```
Dreieck d(-3,0,0,4,3,0);
```

Ausgabe:

```
Konstruktor von (0, 0)
Konstruktor von (0, 0)
Konstruktor von (0, 0)
setXY() (-3, 0)
setXY() (0, 4)
setXY() (3, 0)
```

Einsatzgebiete der Initialisierungsliste

- Die Initialisierungsliste kann verwendet werden zur Initialisierung von nicht-konstanten Attributen.
- Die Initialisierungsliste muss verwendet werden für eingebettete Objekte, für die ein parametrisierter Konstruktor aufgerufen werden soll.
- Weitere Einsatzgebiete kommen im Rahmen der Vererbung hinzu.

Konstante Datenelemente in Objekten (1)

```
class C{
  const int c1, c2;
  int i1, i2;
public:     // Initialisierungsliste des Konstruktors
  C(int n) : c2(n), c1(10), i1(20)
  {
      i2 = n + 30; // Anweisungsblock des
   Konstruktors
  }
  void func(void) {
     i2 = 20;
     c1 = 30; // Fehler !
  }
};
```

Konstante Datenelemente in Objekten (2)

- Konstante Datenelemente können für jedes Objekt verschieden sein.
- Sie werden erst zur Laufzeit festgelegt (z.B. über einen Parameter beim Konstruktor), sind aber ab dann konstant und dürfen nicht mehr geändert werden.
- Die Initialisierung eines konstanten Datenelementes muss(te) über den Konstruktor erfolgen.
- Ab C++/11 können konstanten Datenelemente auch über die **klasseninternen Initialisierer** gesetzt werden
- Dafür gibt es die Initialisierungsliste eines Konstruktors.

Konstante Instanzmethoden (1)

- Eine Instanzmethode, die für ein konstantes Objekt aufgerufen werden soll, muss mit **`const`** zwischen Parameterliste und Funktionsblock deklariert werden:

```
class Bruch {
     int zaehler, nenner;
public:
     ...
     int getZaehler() const;
     void print() const;
     ...
};
```

Konstante Instanzmethoden (2)

- Das Schlüsselwort **const** muss auch bei der Definition der Methode stehen:

```
int Bruch::getZaehler() const {
    return zaehler;
}
void Bruch::setZaehler(int z) const {
    zaehler = z;      // Fehlermeldung da const
}
```

- Eine mit **const** definierte Methode darf weder Datenelemente des this-Objekts verändern noch nicht-const Funktionen für das this-Objekt aufrufen.

Konstante Instanzmethoden (3)

- Methoden, die auf einem **konstanten** Objekt ausgeführt werden, müssen als **const** deklariert sein.
- Methoden, die auf einem **nicht-konstanten** Objekt ausgeführt werden, **können** als **const** deklariert sein.
- Ist sowohl eine const- also auch eine nicht-const-Methode deklariert, so wird bei einem nicht-konstanten Objekt die nicht-const-Methode ausgeführt, bei einem const-Objekt hingegen die const-Methode.

Konstante Parameter in Elementfunktionen

- Ein weiteres Beispiel für konstante Objekte sind mit **`const`** geschützte Parameter:

```
Bruch::Bruch (const Bruch & b){
     cout << "Kopierkonstruktor" << endl;

    // Nur möglich, wenn Methoden const:
     zaehler = b.getZaehler());
     nenner = b.getNenner());
}
```

Konstante Objekte

```
class Bruch {
  int zaehler, nenner;
public:
  Bruch (int zaehler=0 , int nenner =1);
  Bruch (const Bruch & b);
  int getZaehler();
  int getNenner();
  void print();
};

int main() {
  const Bruch g(3,5);
  g.getZaehler();     // Fehler: getZaehler() nicht const
  g.print();          // Fehler: print() nicht const
}
```

Freundfunktionen

- Freundfunktionen einer Klasse sind keine Methoden der Klasse. Sie haben aber die gleichen Zugriffsrechte wie solche.
- Eine Klasse erklärt eine Funktion zur Freundfunktion - und nicht umgekehrt.
- Freundfunktionen können im privaten als auch im öffentlichen Teil der Klasse deklariert werden.
- Freundfunktionen werden nicht wie Instanzmethoden auf Objekten aufgerufen. Man muss ihr ein Objekt der Klasse als Parameter übergeben.
- Freundfunktionen sind nützlich für die Überladung von Operatoren (kommt später).

Beispiel: Freundfunktion (1)

```
class Alpha {

  friend void func100 (Alpha & b);
  friend int getNumber();

  float zahl;
  static int number;

public:
  Alpha(){ zahl = 10.0f; ++number; }

  void print(){ // 2 Stellen nach dem Komma
    cout << fixed << setprecision(2) << zahl << endl;
  }

};
```

Beispiel: Freundfunktion (2)

```
// Hier kommen die Freundfunktionen:
void func100 (Alpha & b){ b.zahl += 100.0f;}

int getNumber(){ return Alpha::number; }

int Alpha::number = 0;

void main(){
  cout << getNumber();  // number == 0
  Alpha a1;
  a1.print();           // zahl == 10.00
  func100(a1);
  a1.print();           // zahl == 110.00
  cout << getNumber();  // number == 1
}
```

Befreundete Klassen

```
class Bruch {
  int zaehler, nenner;
public:
  ..... // beispielsweise keine Methoden

  friend class BruchUtilities;
};

class BruchUtilities {
public:
  static int getZaehler(Bruch & b);
  static void setZaehler(Bruch & b, int z);
  static void print(Bruch & b);
};

void BruchUtilities::setZaehler(Bruch & b, int z)
  { b.zaehler = z; }
```

Objekte als Datenelemente einer Klasse

- Objekte als Datenelemente einer Klasse waren schon zu sehen:

```
class Kreis {.....};
class Doppelkreis {
    Kreis k1, k2;
.....};
```

```
class Punkt {.....);
class Dreieck {
      Punkt p1, p2, p3;
.....);
```

- Zwischen den Objekten besteht eine Teil-Ganzes-Beziehung (part-of).
- Die Teilobjekte werden manchmal auch als eingebettete Objekte bezeichnet.

Beispiel für Objekte als Datenelemente

```
class Kreis {
   double mittelp, radius;
public:
   Kreis (double m, double r) : mittelp(m), radius(r)
       { cout <<     .....};
   ~Kreis () { cout <<     .....};
}

class Doppelkreis {
   Kreis k1, k2;
public:
   Doppelkreis (): k2(1,2), k1(3,4)
       { cout <<     .....};
   ~Doppelkreis () { cout <<     .....};
};
```

Beispiel für Objekte als Datenelemente

```
class Kreis {
   double mittelp, radius;
public:
   Kreis (double m, double r) : mittelp(m), radius(r)
       { cout <<     .....};
   ~Kreis () { cout <<     .....};
}

class Doppelkreis {
   Kreis k1, k2;
public:
   Doppelkreis (): k2(1,2), k1(3,4)
       { cout <<     .....};
   ~Doppelkreis () { cout <<     .....};
};
```

Ausgabe:

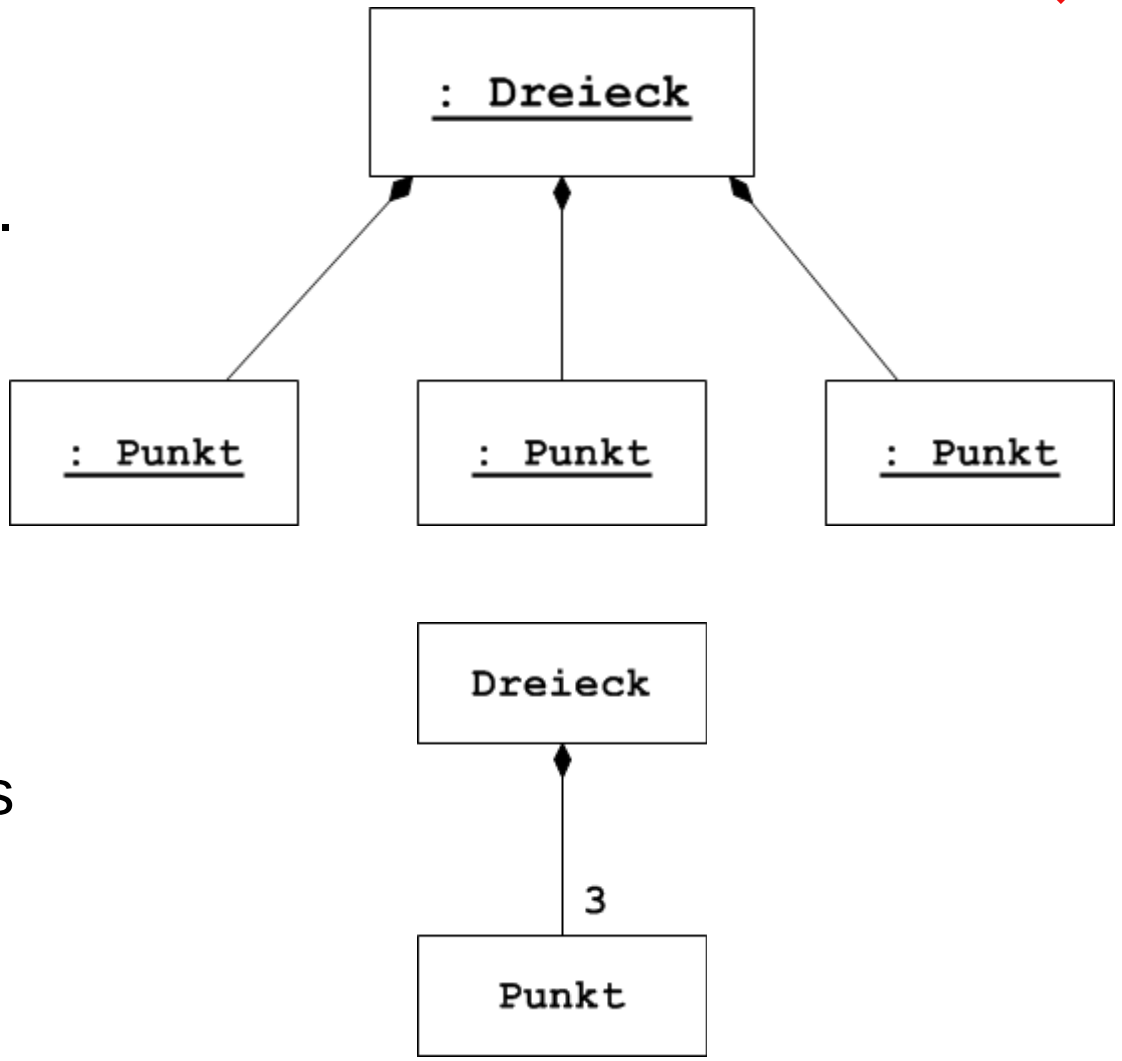
```
Kreis(3.0, 4.0)
Kreis(1.0, 2.0)
Doppelkreis()
~Doppelkreis()
~Kreis(1.0, 2.0)
~Kreis(3.0, 4.0)
```

Komposition vs. Aggregation

- Ein Doppelkreis setzt sich aus Kreisen zusammen! Die Existenz der Kreise ist vom Doppelkreis abhängig! Verschwindet der Doppelkreis, verschwinden auch die Einzelkreise.
- Dies ist ein Beispiel für eine **Komposition** (*part-of*-Beziehung): die Existenz der Teilobjekte ist vom Gesamtobjekt abhängig.
- Eine **Aggregation** ist ebenfalls eine *part-of*-Beziehung. Dabei haben aber die Teilobjekte eine vom Gesamtobjekt unabhängige Existenz.

Beispiel Komposition (Aufgabenstellung)

- Die Klasse **Dreieck** setzt sich aus 3 Objekten der Klasse **Punkt** zusammen.
- Definieren Sie einen Konstruktor mit Parametern, einen Destruktor und die 3 Methoden der Klasse **Dreieck.**
- Die Klasse **Punkt** wird als bekannt vorausgesetzt (siehe nächste Folie).



Beispiel Komposition (Klasse Punkt)

```
// Die Klasse Punkt wird von Dreieck verwendet
class Punkt {
  double x, y;
public:
  Punkt() { x = 0; y = 0; }
  Punkt(double x, double y) {
    this->x = x;  this->y = y;
    cout << "Konstruktor von: " << x << ", " << y << endl; }
  ~Punkt(){ cout << "Destruktor von " << x << ", " << y
            << endl;}
  double getX() const { return x; }
  double getY() const { return y; }
  void setX(double x) { this->x = x; }
  void setY(double y) { this->y = y; }
  void print() {
    cout << "x= " << x << endl;
    cout << "y= " << y << endl; }
};
```

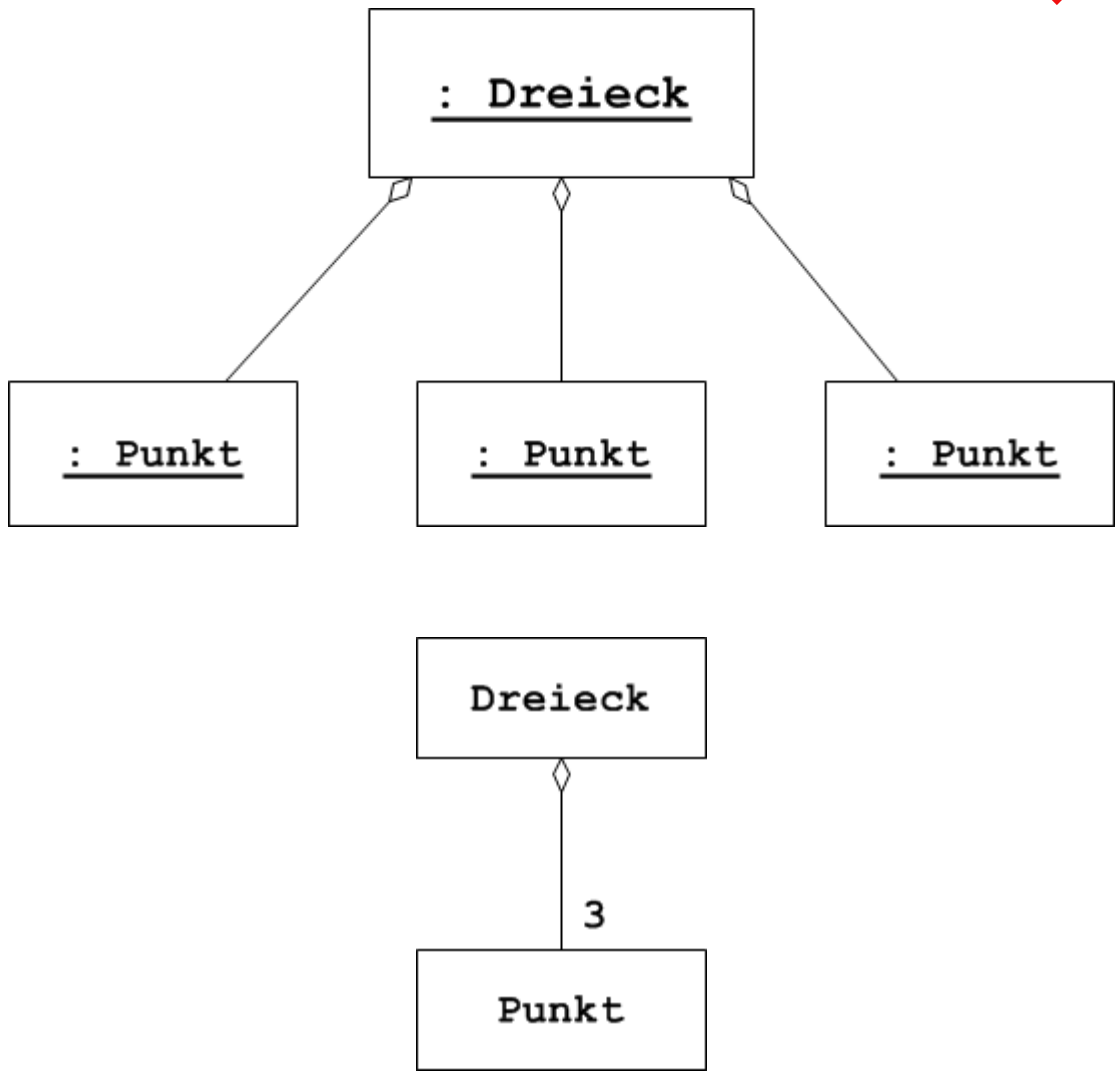
Beispiel Komposition (Klasse Dreieck)

```
// Ein Dreieck HAT 3 Punkte
class Dreieck{
  Punkt p1;
  Punkt p2;
  Punkt p3;
public:
  Dreieck(...):...{...};
  ~Dreieck(){...};
  double umfang() const;
  void print() const; // gibt die Infos zu den 3 Punkten aus
};

void main()
{
  Dreieck d(-3,0,0,4,3,0);
  d.print();
  cout << "Umfang= " << d.umfang() << endl;
}
```

Beispiel Aggregation (Aufgabenstellung)

- Die Klasse **Dreieck** setzt sich aus 3 Objekten der Klasse **Punkt** als *Aggregation* zusammen.
- Definieren Sie einen Konstruktor mit Parametern, einen Destruktor und die 3 Methoden der Klasse **Dreieck.**
- Die Klasse **Punkt** ist wie bisher auch gegeben.



Beispiel Aggregation (Klasse Dreieck)

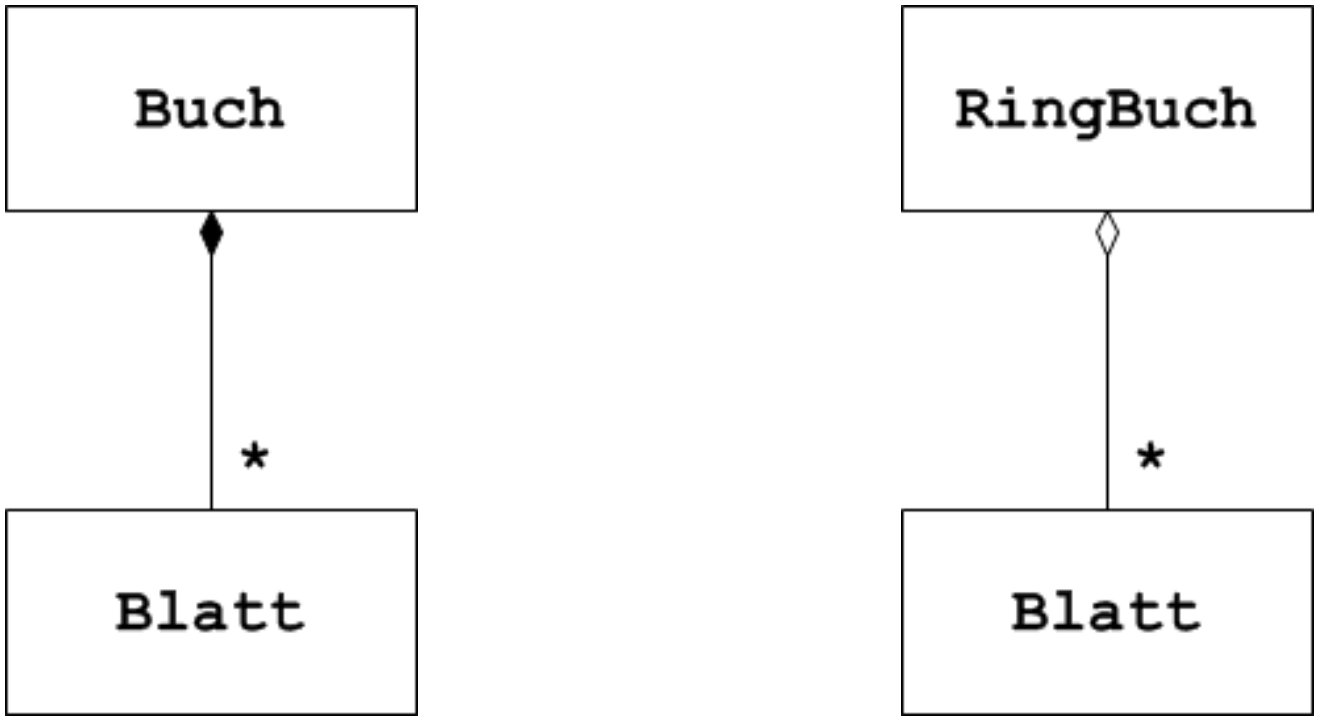
```
// Aggregation: mit Pointern
class Dreieck{
  Punkt *p1;
  Punkt *p2;
  Punkt *p3;
public:
  Dreieck(Punkt * p1, Punkt * p2, Punkt * p2) {...};
  ~Dreieck(){/* loescht die Punkte nicht */ ...};
};

void main()
{
  Punkt pkt1(-3,0), pkt2(0,4), pkt3(3,0);
  // Punkte gehoeren zum Hauptprogramm und nicht zum Dreieck
  Dreieck d(&pkt1, &pkt2, &pkt3);
  d.print();
  cout << "Umfang= " << d.umfang();
}
```

Fazit

- **Komposition:**
  - Die Teilobjekte bilden ein unteilbares Ganzes.
  - Die Teilobjekte haben keine unabhängige Existenz.
  - Die Teilobjekte sind als Attribute eingebettet.
- **Aggregation:**
  - Die Teilobjekte können isoliert betrachtet werden.
  - Die Teilobjekte haben eine unabhängige Existenz.
  - Die Teilobjekte werden über Pointer oder Referenzen eingebettet.

Weiteres Beispiel



Zusammenfassung

- Statische Klassenelemente (Datenelemente, Funktionen) gehören zu der Klasse und nicht zu den Objekten.
- Klassenvariablen müssen außerhalb der Klasse initialisiert werden.
- Konstante Datenelemente in Objekten werden einmal im Konstruktor (Initialisierungsliste) gesetzt und können dann nicht mehr geändert werden.
- Konstante Instanzfunktionen werden benötigt, wenn man auf konstante Objekte zugreifen will.
- Freundfunktionen, Freundklassen können auch auf private Elemente einer Klasse zugreifen. Die Klasse muss aber seine Freunde selber aussuchen.

Zusammenfassung

- Eingebettete Objekte sind Attribute einer Klasse vom Typ einer anderen Klasse.
- Eingebettete Objekte werden erzeugt, indem im eigenen Konstruktor der Konstruktor der anderen Klasse aufgerufen wird.
- Soll ein parametrisierter Konstruktor aufgerufen werden, kann dies in der Initialisierungsliste erfolgen.
- Die Konstruktoren werden in der Reihenfolge aufgerufen, wie die Objekte in der Klasse deklariert werden. Die Reihenfolge der Initialisierungsliste spielt keine Rolle.
- Die eingebetteten Objekte werden im Destruktor auch wieder gelöscht - in der umgekehrten Reihenfolge, wie sie erzeugt werden.

Zusammenfassung

- Aggregation und Komposition sind Teil-Ganzes-Beziehungen (part-of) zwischen Objekten.
- Bei einer Komposition kontrolliert das Gesamtobjekt die Teilobjekte. Die Teilobjekte haben keine Existenz außerhalb des Gesamtobjektes.
- Bei einer Aggregation gehen Gesamtobjekt und Teilobjekte die Verbindung nur für eine bestimmte Zeit ein. Die Teilobjekte können unabhängig vom Gesamtobjekt existieren.
- Bei eingebetteten Objekten handelt es sich um eine Komposition.
- Bei einer Aggregation werden die Teilobjekte über Pointer oder Referenzen aus dem Gesamtobjekt angesprochen.

Exkurs: Konstruktor-Delegation in C++11

- Bisher: Ein Konstruktor kann nicht einen anderen Konstruktor aufrufen. Denn wenn ein Konstruktor seine Arbeit beendet hat, ist das Objekt fertig.

```
class Bruch {…
Bruch (int z, int n) {zaehler = z; nenner = n;}
Bruch () {Bruch(1,1);}  // nur ein temp. Objekt
…
}
```

- Wenn man viele und vielfältig parametrisierte Konstruktoren in einer Klasse hat, gibt es viele Codeduplikate. Änderungen können nicht an einer Stelle durchgeführt werden.

Exkurs: Konstruktor-Delegation in C++11

- In C++11 ist jetzt eine Konstruktor-Delegation möglich.
- Dies geschieht in der Initialisierungsliste:

```
class Bruch {…
Bruch (int z, int n) {zaehler = z; nenner = n;}
Bruch () : Bruch(1,1) {}  // Delegation
…
}
```

- Die Initialisierungsliste darf dann nur aus dieser Delegation bestehen.
- Es gibt dabei aber jede Menge Probleme zu beachten, insbesondere wenn im Konstruktor Exceptions auftreten.

Exkurs: Attribute initialisieren in C++11

- C++11 erlaubt eine neue Schreibweise in der Klassendefinition, um die Initialisierung zu vereinfachen:

```
class Bruch {
    int zaehler = 0;
    int nenner = 1;
public:
    Bruch() {}
    Bruch (int z): zaehler(z) {}
    Bruch (int z, int n): zaehler(z), nenner(n) {}
…
}
```

Exkurs: Attribute initialisieren in C++11

- Diese Initialisierung wird ausgeführt, wenn der Konstruktor keine Werte für die Attribute vorsieht.
- Es entsteht keine doppelte Initialisierung, daher darf diese Schreibweise auch für const-Attribute verwendet werden.
- Diese Werte gelten für alle Konstruktoren, wodurch man einiges an Code sparen kann.
- Diese Schreibweise zur Initialisierung kann auch bei Klassenvariablen verwendet werden, allerdings nur bei einfachen Typen.