C++ Teil 10

Sven Groß

21. Dez 2015

Themen der letzten Vorlesung

Objektorientierung und Klassen

- Strukturen (Structs)
- Klassen
- Überladen von Operatoren
- Attribute und Methoden

Heutige Themen

Objektorientierung und Klassen

1. Attribute und Methoden
2. Konstruktor und Destruktor
3. Zugriffskontrolle

Klassen: Attribute und Methoden (Wdh.)

- **Objektorientierung**: Objekte beinhalten nicht nur **Daten**, sondern auch **Funktionalität** des neu definierten Datentyps
- Konzeptionelle Erweiterung von Structs: **Klassen**
- Elemente einer **Klasse**:
  - **Attribute** oder **Datenelemente**: Daten (vgl. Structs)
  - **Methoden** oder **Elementfunktionen**: Funktionen, die das Verhalten der Klasse beschreiben und auf Datenelementen operieren (auch in Structs)

```
class Student
{
  public:
    string Name;                    // Datenelemente
    int    MatNr;
    double Note;

    bool hat_bestanden() const; // Methoden
    void berechne_Note( int Punkte);
}; // Semikolon nicht vergessen !!!
```

Klassen und Objekte (Wdh.)

```
Student s; // erzeugt Variable s vom Typ Student

s.Name= "Hans␣Schlauberger";
s.Note= 1.3;
s.MatNr= 234567;

if ( s.hat_bestanden() )
{
  cout << "Herzlichen␣Glueckwunsch!" << endl;
}
```

- Datentyp `Student` heisst auch **Klasse**
- Variable `s` heisst auch **Objekt**
- bereits bekannte Klassen:
  `string, vector<...>, ifstream, ...`

Methodendefinition (Wdh.)

- entweder *inline* innerhalb der Klassendefinition (`student.h`), nur für kurze Methoden sinnvoll

```
class Student
{
  ...
    bool hat_bestanden() const
    {
      return Note <= 4.0;
    }
  ...
};
```

- oder Deklaration in Klasse, Definition außerhalb (`student.cpp`)

```
bool Student::hat_bestanden() const
{
  return Note <= 4.0;
}
```

const-Qualifizierung

```
class Student
{
  ...
    bool hat_bestanden() const; // Methoden
    void berechne_Note( int Punkte);
};
```

- `const` hinter Methode sichert zu, dass diese das Objekt nicht verändert
- `berechne_Note` kann nicht `const` sein, da Attribut `Note` verändert wird

```
const Student bob;          // bob ist konstant

if (bob.hat_bestanden())    // ok
  cout << "Glueckwunsch!"

bob.berechne_Note( 34);     // Compiler-Fehler, fuer bob
                            // duerfen nur const-Methoden
                            // aufgerufen werden
```

Objektzeiger

- Wdh.: Zugriff auf Attribute und Methoden über Objektzeiger:

```
Student s;
Student *s_zeiger= &s;

(*s_zeiger).Note= 3.3;
if ( (*s_zeiger).hat_bestanden() ) ...
```

einfachere Schreibweise mit `->` ist besser lesbar:

```
s_zeiger->Note= 3.3;
if ( s_zeiger->hat_bestanden() ) ...
```

- in jeder Methode wird implizit ein `this`-**Zeiger** übergeben, der auf das aufrufende Objekt zeigt

```
void Student::setzeName( string Name)
{
    this->Name= Name;
}
```

this-Zeiger – Beispiel

Beispiel: Zuweisungsoperator `operator=` als Methode

- Definition:

```
Student& Student::operator= ( const Student& s)
{
    Name= s.Name;
    this->MatNr= s.MatNr;
    Note= s.Note;
    return *this;    // gibt sich selber zurueck
}
```

- Aufruf:

```
Student bobby;

bobby= bob;  // Zuweisung
```

  - Zuweisung `bobby= bob` bewirkt Aufruf `bobby.operator=( bob )`
  - `this` zeigt auf `bobby`

Konstruktor

- wird beim Erzeugen eines Objektes automatisch aufgerufen
- Zweck: initialisiert Attribute des neuen Objektes
- Klasse kann mehrere Konstruktoren haben:
  - Default-Konstruktor
    ```
    Student() { ... }
    ```
  - Kopierkonstruktor
    ```
    Student( const Student& s) { ... }
    ```
  - darüber hinaus weitere, allgemeine Konstruktoren möglich, z.B.
    ```
    Student (string name, int matnr, double note) { ... }
    ```

```
Student a;                    // per Default-Konstruktor
Student b( "Hans␣Schlau", 234567, 1.3);
Student c( b);                // per Kopierkonstruktor
```

- **Quiz:** Warum darf beim Kopierkonstruktor *kein Wertparameter* benutzt werden?
  ```
  Student( Student s); //nicht erlaubt
  ```

Konstruktor – Beispiel

```
class Student
{ ...
  public:
    Student();
    Student( string name, int matnr, double note);
    Student( const Student&);
};
```

```
Student::Student()
{ // Default-Konstruktor
    Name="Max␣Mustermann";  MatNr= 0;  Note= 5.0;
}

Student::Student( const Student& s)
{ // Kopier-Konstruktor
    Name= s.Name;  MatNr= s.MatNr;  Note= s.Note;
}

Student::Student( string name, int matnr, double note)
{ // weiterer Konstruktor
    Name= name;  MatNr= matnr;  Note= note;
}
```

Objekte erzeugen durch verschiedene Konstruktoren

```
Student alice,  // Default-Konstruktor
        bob( "Bob Meier", 123456, 2.3),
        chris( "Chris Schmitz", 333333, 5.0);

Student lisa( alice); // Kopierkonstruktor
Student bobby= bob;   // Kopierkonstruktor
```

- `Student bobby( bob);` bzw.
  `Student bobby= bob;` synonyme Syntax
  → Kopierkonstruktor
- `Student alice;` → Default-Konstruktor

⚠ `Student alice();` keine gültige Syntax für Default-Konstruktor-Aufruf
(wäre verwechselbar mit Funktionsdeklaration!)

Destruktor

- wird automatisch beim Vernichten eines Objektes aufgerufen
- Zweck: z.B. Speicher eines dynamischen Feldes wieder freigeben
- jede Klasse hat genau *einen* Destruktor
  `~Student () { ... }`
- Jede Klasse besitzt folgende **impliziten** Methoden,
  falls diese nicht anders deklariert sind:
  - Destruktor, der nichts tut (wie `~Student() {}`)
  - Default-Konstruktor (falls keine Konstruktoren deklariert werden)
  - Kopierkonstruktor
  - Zuweisungsoperator `operator=`

```
class Array
{
    double* feld;
  public:
    Array( int laenge) { feld= new double[laenge]; }
    ~Array()           { delete[] feld; }
}; // Problem: impliziter Kop.konstr. und operator= ungeeignet
```

Zugriffskontrolle

- `private`-Bereich: Zugriff nur durch Methoden derselben Klasse
- `public`-Bereich: Zugriff uneingeschränkt über Objekt möglich

```
class Student
{
  private:
    string Name;                    // Attribute
    int    MatNr;
    double Note;

  public:
    // Konstruktor
    Student( string Name, double Note, int MatNr);
    // Destruktor
    ~Student();

    bool hat_bestanden() const; // Methoden
    void berechne_Note( int Punkte);
};
```

Zugriffskontrolle: Klassen vs. Strukturen

- `public`-/`private`-Bereiche auch in Strukturen möglich
- einziger Unterschied: Default-Verhalten, wenn nichts angegeben
  - Strukturen: standardmäßig `public`
  - Klassen: standardmäßig `private`

⇝ für objektorientierte Programmierung werden i.d.R. Klassen verwendet

```
struct Datum
{
    int Tag, Monat, Jahr;    // alles public
  private:
    double geheim;
};

class Date
{
    int Day, Month, Year;    // alles private
  public:
    int GetMonth() const;
    int SetMonth( int month);
};
```

Zugriffskontrolle

- `private`-Bereich: Zugriff nur durch Methoden derselben Klasse
  - in der Regel für Attribute einer Klasse

  ⇝ Attribute sollen vor willkürlichem Zugriff geschützt werden
  - Zugriff auf private Attribute nur über public-Methoden der Klasse möglich
- `public`-Bereich: Zugriff uneingeschränkt über Objekt möglich
  - in der Regel für Methoden einer Klasse

  ⇝ ermöglicht Arbeiten mit einer Klasse über wohldefinierte Schnittstellen

⇝ Vorteile objektorientierter Programmierung:

- **Datenkapselung**: für den Nutzer einer Klasse tritt deren interne Arbeitsweise in den Hintergrund
- **einfache Wartung**: private Interna der Klasse können ausgetauscht werden ohne Auswirkung auf den Nutzer
- **Wiederverwendbarkeit** durch wohldefinierte öffentliche Schnittstelle