C++ Teil 9

Sven Groß

14. Dez 2015

Themen der letzten Vorlesung

- STL: Sequentielle Container
  - Vektoren
  - Listen
- Typedefs und Makros
- File Streams: Ein- und Ausgabe mit Dateien
- Zusammengesetzte Datentypen
  - Strukturen (Structs)

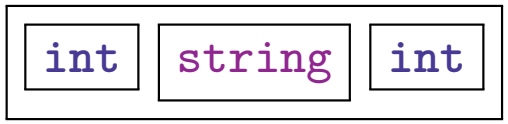
Heutige Themen

Objektorientierung und Klassen

1. Zusammengesetzte Datentypen
   - Strukturen (Structs)
   - Überladen von Operatoren
   - Klassen
2. Attribute und Methoden
3. Konstruktor und Destruktor

Strukturen (*Structs*)

- **Strukturen**: Zusammenfassung unterschiedlicher Datentypen zu einem *neuen benutzerdefinierten Datentyp*
- z.B. neuer Datentyp `Datum`:




```
struct Datum
{
    int    Tag;
    string Monat;
    int    Jahr;
}; // Semikolon nicht vergessen!
```

```
Datum MamasGeb, MeinGeb;

MamasGeb.Tag=   12;
MamasGeb.Monat= "Februar";
MamasGeb.Jahr=  1954;

MeinGeb= MamasGeb;  // Zuweisung
MeinGeb.Jahr+= 30;  // ich bin genau 30 Jahre juenger
```

Objekte

- Struktur-Variablen nennt man auch **Objekte**:
  `MeinGeb` ist ein Objekt vom Datentyp `Datum`.
- Zugriff auf **Datenelemente** eines Objektes mit '`.`' :
  `MeinGeb.Monat` ist ein Datenelement des Objektes `MeinGeb`.

```
cout << "Mein␣Geburtstag␣ist␣im␣" << MeinGeb.Monat;
```

Datenelemente nennt man synonym auch **Attribute**.

- Zugriff auf Datenelemente über Objektzeiger:

```
Datum *datzeig= &MeinGeb;

(*datzeig).Tag= 7;
cout << "Mein␣Geburtstag␣ist␣im␣" << (*datzeig).Monat;
```

einfachere Schreibweise mit `->` ist besser lesbar:

```
datzeig->Tag= 7;
cout << "Mein␣Geburtstag␣ist␣im␣" << datzeig->Monat;
```

Überladen von Operatoren (*operator overloading*)

- praktisch wäre, folgendes schreiben zu können:

```
if (MamasGeb < MeinGeb) // Datumsvergleich
    cout << "Mama ist aelter als ich.";
```

- möglich, wenn man folgende Funktion `operator<` definiert:

```
bool operator< (const Datum& a, const Datum& b)
{
    ...
}
```

Das nennt man **Operatorüberladung**.

- `MamasGeb < MeinGeb` ist dann äquivalent zum Funktionsaufruf `operator< ( MamasGeb, MeinGeb )`
- für viele Operatoren möglich, z.B. `+ - * / == >> <<` usw., je nach konkretem Datentyp sinnvoll

Überladen von Operatoren – Beispiel

Beispiel: Ein-/Ausgabeoperator für `Datum`:

- Wir wollen Code schreiben wie

```
ifstream ifs( "Datum.txt" );    // 3. Dezember 2014
Datum heute;
ifs >> heute;
cout << "Heute␣ist␣der␣" << heute << endl;
```

- Definition des **Ausgabeoperators** `<<` :

```
ostream& operator<< ( ostream& out, const Datum& d)
{
    out << d.Tag << ".␣" << d.Monat << "␣" << d.Jahr;
    return out;
}
```

- **Quiz:** Wie könnte der **Eingabeoperator** `>>` definiert werden?

```
istream& operator>> ( istream& in, Datum& d)
{
    // ???
}
```

Klassen: Attribute und Methoden

- **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

```
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

- 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;
}
```

Destruktor

- wird automatisch beim Vernichten eines Objektes aufgerufen
- Zweck: z.B. Speicher eines dynamischen Feldes wieder freigeben
- jede Klasse hat genau *einen* Destruktor
  `~Student () { ... }`
- wird kein Destruktor deklariert, so gibt es immer einen *impliziten* Destruktor, der nichts tut (wie `~Student() {}`)
- Ebenso gibt es in jeder Klasse auch einen *impliziten*
  - 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
```