C++ Klassen – weitere Funktionen

Übersicht

- static Elemente
- const Elemente
- Zusätzliches zu Konstruktoren
  - Kopier-Konstruktor
  - Konvertierung-Konstruktor
  - Initialisierungslisten
- Friend Funktionen

Statische Klassenkomponenten

- Alle Objekte einer Klasse erhalten bei der Instanzierung ihre eigenen Member-Variablen. Alternativ dazu können statische (static) Member-Variablen deklariert werden, die dann für alle Objekte der Klasse gemeinsam zur Verfügung stehen. Gründe für die Verwendung können sein:
  - Ein Zähler, der die Anzahl der gerade aktiven Objekte der Klasse wiederspiegelt.
  - Ein Zeiger auf einen dynamisch allokierten Speicherbereich, der von allen Objekten der Klasse geteilt wird.
  - Eine Testvariable, die objektübergreifend gemeinsame Informationen aufnehmen kann.

- Vorteile gegenüber der Verwendung globaler Variablen sind:
  - Durch die private-Deklaration kann der Zugriff auf Member-Funktionen beschränkt werden.
  - Der Bezugsrahmen bleibt innerhalb der Klasse. Damit können keine Kollisionen mit anderen Variablen gleichen Namens entstehen.

Statische Member-Variablen

- Regeln für die Initialisierung von static-Member-Variablen:
  - Die Initialisierung darf nur außerhalb der Klassen-Definition vorgenommen werden und wird vom Compiler ohne Instanzierung von Objekten durchgeführt.
  - Bei der Initialisierung muss der Datentyp angegeben und der Klassenbezug hergestellt werden.
  - Das Schlüsselwort static darf nur bei der Definition und nicht bei der Initialisierung stehen.
  - Die Variable kann initialisiert werden, auch wenn sie in der Klasse als private deklariert wurde.
  - Eine Variable darf nur einmal initialisiert werden.
  - Wird die Variable nur definiert und nicht initialisiert, so wird sie vom Compiler vorbelegt.

Statische Member-Variablen

Beispiel für die Initialisierung von static-Member-Variablen:

```
class Demo
{
private:
    static double stv;          //Initialisierung hier nicht möglich
};

.......
double Demo::stv = 10.;         //Initialisierung trotz private
Demo::stv = 10.:                //Fehler: Typangabe fehlt
static double Demo::stv = 10.;  //Fehler: static nicht erlaubt
double stv = 10.;               //Fehler: Klassenbezug notwendig
double Demo::stv = 20.;         //Fehler: Variable bereits initialisiert
```

Empfehlung:

Initialisierung von static-Member-Variablen nicht in Header-Dateien vornehmen, sondern zusammen mit der Definition der Member-Funktionen.

Statische Member-Variablen

Regeln für den Zugriff auf static-Member-Variablen:

- *Elementfunktionen* der Klasse haben Zugriff auf static-Member-Variablen wie auf alle anderen Member-Variablen.
- *Objekte* der Klasse haben Zugriff auf static-Member-Variablen wie auf alle anderen Member-Variablen incl. aller Einschränkungen (Zugriffsschutz).
- Auf die static-Member-Variablen kann *direkt* unter Verwendung des Bezugs-operators und des Klassennamens zugegriffen werden. Allerdings greift auch hier der Zugriffsschutz.

Empfehlung:

Greifen Sie auf static-Member-Variablen *nicht* über die Instanzierung von Objekten sondern besser *direkt* zu, da die Variable allen Objekten *gemeinsam* zur Verfügung steht.

Statische Member-Variablen

Beispiel für den Zugriff auf static-Member-Variablen:

```
class Demo
{
private:
    static int stvint;
public:
    static double stvdbl;
};

.......
int Demo::stvint = 10;              //Initialisierung trotz private
double Demo::stvdbl = 20.;          //OK
int j = Demo::stvint;               //Fehler: Zugriffsverletzung
Demo::stvint = 100;                 //Fehler: Zugriffsverletzung
double x = Demo::stvdbl;            //OK
Demo::stvdbl = 200.;                //OK
Demo doobj;                         //Instanziiere Objekt
int k = doobj.stvint;               //Fehler: Zugriffsverletzung
doobj.stvint = 100;                 //Fehler: Zugriffsverletzung
double y = doobj.stvdbl;            //erlaubt, aber nicht empfohlen
doobj.stvdbl = 200.;                //erlaubt, aber nicht empfohlen
```

Statische Member-Funktionen

Problem:

- Eine Variable, die sowohl static als auch private deklariert ist, kann zwar direkt initialisiert werden, aber für den sonstigen Zugriff über eine gewöhnliche Element-funktion muss immer ein Objekt instanziert werden. Nach den Ausführungen über static-Member-Variablen sollte genau dies aber vermieden werden.

```
class Demo
   {
   private:
     static int stv;
   public:
     int GetNumber(void)
          {return stv;}

      //inline-Definition
   };

.......
   Demo doobj;

   //Instanziiere Objekt
   int j = doobj.GetNumber( );

   //Der einzige Weg zum Datum
```

Statische Member-Funktionen

Lösung:

Deklariere eine static-Member-Funktion, die folgende Eigenschaften aufweist:

- Die static-Member-Funktion wird definiert wie eine normale Element-Funktion.
- Die static-Member-Funktion kann *direkt* unter Verwendung des Bezugs-operators und des Klassennamens auf die static-Member-Variablen zugreifen, *ohne* ein Objekt instanzieren zu müssen. Allerdings greift auch hier der Zugriffsschutz.
- Der Compiler ergänzt *keinen* this-Zeiger. Daher kann die static-Member-Funktion ohne weitere Maßnahmen *nur* auf static-Member-Variablen zugreifen. Für Nicht-static-Member-Variablen *müssen* Objekte instanziert werden.
- Der Aufruf als normale Funktion über die Instanzierung von Objekten ist möglich, sollte jedoch vermieden werden.

Beispiel: Statische Member-Funktionen

```
class Demo
{
private:
    double nstvdbl;                          //non-static-Variable
    static int stvint;                       //static-Variable
    static void SetNumber(int i, double x);        //Deklaration
public:
    static int GetNumber(void) {return stvint;};   //inline-Definition
};

void Demo::SetNumber(int i, double x)  //Definition
{
    stvint = i;                          //OK: static
    nstvdbl = x;                         //Fehler: non-static
}
.......
Demo::SetNumber(2, 3.);                  //Fehler: Zugriffsverletzung
int j = Demo::GetNumber( );              //OK: direkter Zugriff
Demo doobj;                              //Instanziiere Objekt
int k = doobj.GetNumber( );              //erlaubt, aber nicht empfohlen
```

Konstante Klassenkomponenten

Bei der Instanzierung eines Objektes einer Klasse kann diesem die Eigenschaft const zugeordnet werden. Damit darf das Objekt die Datenelemente seiner Klasse weder direkt (soweit public) noch indirekt über seine Elementfunktionen modifizieren. Für den lesenden Zugriff auf die Datenelemente muss die Elementfunktion ebenfalls als const deklariert sein.

Regeln für const-deklarierte Elementfunktionen:

- Eine const-deklarierte Elementfunktion darf nur von const-deklarierten Objekten einer Klasse aufgerufen werden.
- Das Schlüsselwort const muss sowohl bei der Deklaration als auch bei der Definition angegeben werden.
- Eine const-deklarierte Elementfunktion kann von einer nicht-const-deklarierten Elementfunktion gleichen Namens überladen werden.

Empfehlung:

Verwenden Sie const-deklarierte Objekte insbesondere dort, wo große Objekte zur Einsparung von Kopieraufwand beim call-by-value als Zeiger oder Referenz übergeben werden.

Beispiel für const-deklarierte Elementfunktionen:

```
class Demo
{
private:
     int i;
     double *ptdbl;                          //Zeiger
public:
     int GetNumber(void) const;              //Deklaration
     void SetNumber1(double x) const;        //const-Elementfunktion
     void SetNumber1(double x);             //überladene Elementfunktion
     void SetNumber2(double *y) const;       //Deklaration
};

void Demo::SetNumber1(double x) const        //Definition
     {*ptdbl = x;}                          //OK: Zeiger bleibt unverändert

void Demo::SetNumber1(double x)              //überladene Elementfunktion
     {*ptdbl = x;  i = 10}                   //OK: Funktion darf alles

void Demo::SetNumber2(double *y) const       //Definition
     {ptdbl = y;}                            //Fehler: Zeiger darf nicht
                                             //verändert werden

.......
const Demo conobj;                           //const-Objekt
Demo varobj;
conobj.SetNumber1(10.);                      //verwendet SetNumber1 const
varobj.SetNumber1(20.);                      //verwendet SetNumber1 non-const
```

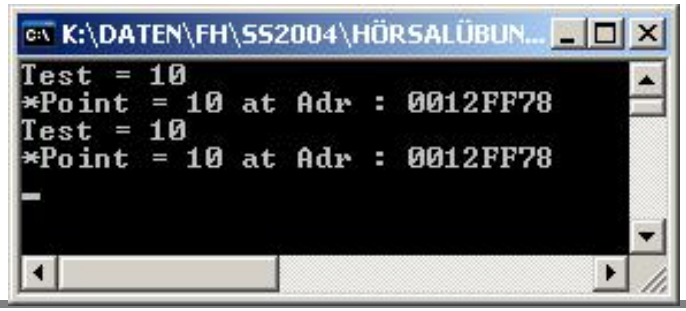
Konstruktoren

- Spezielle Konstruktoren
  - Standard Konstruktor (siehe vorh. Abschnitt)
    - KlassenName (void) {..}
    - KlassenName (int k=0) {..}
  - Kopier-Konstruktor
    - KlassenName (const KlassenName &Objekt)
  - Konversions-Konstruktor
    - KlassenName (const AndereKlasse &Objekt)

Kopier-Konstruktor

- Wird ein Objekt bei der Instanzierung durch ein anderes Objekt der gleichen Klasse initialisiert, und fehlt die explizite Definition eines Kopier-Konstruktors, so generiert der Compiler einen einfachen Kopier-Konstruktor, der die Inhalte der Member-Variablen elementweise kopiert:
- Problem:
  - Beide Objekte haben zwar unterschiedliche Zeiger aber sie zeigen auf die identische Speicheradresse.

```
class Demo
{ private:
    int Test;
    int *Point;
  public:
    Demo() {Test=10; Point=&Test;}
    void print ()
    { cout << "Test = " << Test << endl
           << "*Point = " << *Point
           << " at Adr : "
           << Point << endl; }
};
int main ()
{   Demo obj;
    Demo obj1(obj);
    obj.print();
    obj1.print();
    return 0;}
```



Kopier-Konstruktor

- Führt eine "flache" Kopie aller Variablen (Standard Kopier-Konstruktor) zu falschen Ergebnissen:
  - Eigenen Kopier-Konstruktor schreiben
- Eigenschaften
  - genau ein Referenzparameter derselben Klasse normalerweise const
  - Evtl. noch zusätzliche Default-Parameter
  - kein normaler Parameter derselben Klasse
    - da call by value sonst zum rekursiven Aufruf führt

```
class Demo
    {
    private:
    int elemente;
    double *vektor;
    public:
      Demo(void); //Standard-Konst.
      Demo(int k);//überl.Konstruktor
      Demo(const Demo &obj);
                 //Kopier-Konstruktors
    };
    Demo::Demo(const Demo &obj) {
      elemente = obj.elemente;
      vektor = new double [elemente];
      for (int i = 0, i < elemente,
           i++)
        vektor[i] = obj.vektor[i];
    }

.......
    Demo obj1(10);     // Demo(int)
    Demo obj2 = obj1; // Kopie (obj1)
    Demo obj3(obj2);  // Kopie (obj2)
```

Zusammenfassung Kopier-Konstruktor

- Der automatische Aufruf des Kopier-Konstruktors erfolgt, wenn
  - ein Objekt einer Klasse instanziert und mit einem Objekt der gleichen Klasse initialisiert wird oder
  - ein Objekt als Aktualparameter einer Funktion verwendet (call-by-value) wird oder
  - ein Objekt Rückgabewert (call-by-value) einer Funktion ist.
- Ist der Kopier-Konstruktor nicht explizit definiert, so wird er vom Compiler generiert.
- Formen des Kopier-Konstruktors, wenn Demo die Klasse ist und obj ihr Objekt:
  - Demo (const Demo &obj);
  - Demo (Demo &obj);
  - Demo (const Demo &obj, int i = 0, double x = 0.);
  - Demo (Demo &obj, int i = 0, double x = 0.);
- Kopier-Konstruktor immer mit Referenzparameter (->Rekursiv)
- Falls man das Kopieren von Objekten geziehlt verhindern möchte
  - Demo (Demo &obj) ; // deklarieren aber nicht definieren

Konversions-Konstruktor

- Wird ein Objekt bei der Instanzierung durch ein anderes Objekt einer anderen Klasse initialisiert, so bedarf es eines Konversions-Konstruktors.
  - Ein solcher Konstruktor wird vom Compiler nicht automatisch generiert.
  - Das Fehlen des Konversions-Konstruktors wird vom Compiler moniert.

```
class Klasse
    {
    .........
    };

class Demo
    {
     public:
          Demo(const Klasse &pp);
    //Deklaration des Konversions-
    Konstruktors  .........
    };

..........
    Klasse objKlasse(100);
    //Instanziiere + initialisiere
    Objekt aus Klasse


    Demo  objDemo(objKlasse);
    //Instanziiere Objekt aus Demo
    //und initialisiere mit objKlasse
```

Initialisierungslisten

- Bisher wiesen die Member-Variablen von Klassen nur einfache Datentypen auf. Sind Objekte einer anderen Klasse Member-Variablen der zu definierenden Klasse, so werden diese mit dem Konstruktor der zu definierenden Klasse über Initialisierungslisten initialisiert.

```
class Sample
    {
    public:
      Sample(Sample &ss);// Kopier-K.
    };
    // Sample muss definiert sein!
    class Demo
    {
    private:
      Sample objSample;
      const double y;
      int k;
    public:
      Demo(int i, double x, const
           Sample &oo);
    };


Demo::Demo (int i, double x,
             const Sample &oo)
      : y(x), objSample(oo)
    {k = i;}

…

Sample obj;

Demo oDemo(1, 1.4, obj);
```

Initialisierungslisten

Syntax der Initialisierungsliste:

```
klassenname::klassenname(parameterliste) : initialisierungsliste
```

Eigenschaften der Initialisierungsliste:

- Sie steht bei der *Definition* des Konstruktors, *nicht* bei der Deklaration.
- Bei einfachen Variablen wird der Wert in der Klammer zur Initialisierung der Variablen verwendet. Die Klammer kann auch einen komplexen Ausdruck ent-halten.
- Bei Objekten wird der Kopier-Konstruktor zur Initialisierung des Objektes verwendet. Hat die Klasse, zu der das Objekt gehört, *keinen* Kopier-Konstruk-tor, so wird einer vom Compiler generiert, der eine *einfache* Kopie der Daten durchführt.
  Empfehlung: Definieren Sie *immer* einen Kopier-Konstruktor.
- Variablen und Objekte erhalten ihre Werte aus der Initialisierungsliste, *bevor* der Konstruktor zur Ausführung kommt.
- Die Member-Variablen werden in der Reihenfolge initialisiert, wie sie in der Definition der Klasse aufgeführt sind, *nicht* in der Reihenfolge, wie sie in der Initialisierungsliste erscheinen.
- Normale Variablen *können* durch Initialisierung *oder* durch Zuweisung mit Werten versorgt werden. Objekte und const-Variablen *müssen* initialisiert wer-den.

friend-Klassenkomponenten

Ergänzend zu dem bisherigen Zugriffschutz für das Innere von Objekten mit zwei Differenzierungen

- private (*keiner* darf von außen zugreifen, *außer* über Elementfunktionen)
- public (*alle* dürfen von außen zugreifen)

kann es sinnvoll sein, besonders „*vertrauenswürdigen*“ Elementen den freien Zugriff auf private-Elemente zu gestatten. Solche Elemente können sein:

- globale Funktionen
- Elementfunktionen *anderer* Klassen
- ganze Klassen

Syntax:
- friend typ funktionsname(parameter);
- friend class klassenname;
- friend typ klassenname::elementfunktionsname(parameter);

Empfehlung:

Gehen Sie zurückhaltend mit dem Einsatz des friend-Konzepts um, da hiermit indirekt der Zugriffsschutz (Datenkonsistenz) umgangen wird.

friend-Klassenkomponenten

- Die friend-Deklaration erfolgt innerhalb der Definition derjenigen Klasse, deren private-Daten und private-Elementfunktionen zum Zugriff freigegeben werden sollen.

- Die Deklaration der friend-Klasse oder friend-Funktion kann irgendwo innerhalb der Klassendefinition stehen.
  - Empfehlung: Direkt hinter den Deklarationskopf der Klasse positionieren.

```
class Klasse1;//Vorwärts-Deklaration
class Klasse2;//Vorwärts-Deklaration

class Demo
{
   friend class Klasse1; //Klasse
   private:
     friend void Funktion1(float x);
     //Funktion1 ist friend zu Demo
     //und globale Funktion

   public:
     friend int Klasse2::Funktion2
                          (int i);
     //Funktion2 ist friend zu Demo
     //und Elementfunkt. von Klasse2

};
```

friend-Klassenkomponenten

- Ist eine *ganze* Klasse (B) friend-Klasse einer anderen Klasse (A), dann können *alle* Elementfunktionen der friend-Klasse (B) auf die private-Daten und private-Elementfunktionen von Klasse (A) zugreifen. Umgekehrt gilt dies *nicht*.
- Ist die Klasse, die friend-Klasse einer zu definierenden Klasse werden soll, bisher selbst noch nicht definiert, so ist wenigstens eine Vorwärts-Deklaration notwendig und ausreichend.
- Eine Funktion, die in einer Klasse als friend-Funktion deklariert wurde, ist *nicht* Member-Funktion dieser Klasse, d.h.
  - die friend-Funktion befindet sich *nicht* im Bezugsrahmen der Klasse
  - die friend-Funktion selbst hat *keinen* direkten Zugriff auf die Member-Daten. Es sind die instanzierten *Objekte* der Klasse, die innerhalb der Definition der friend-Funktion den Zugriff auf die Member-Daten ermöglichen.
  - der Klassenbezugsoperator darf bei der Definition der Funktion *nicht* verwendet werden.
- Eine friend-Funktion kann mit einer anderen Funktion überladen werden. Falls alle Funktionen friend-Funktion der Klasse sein sollen, so muss *jede* überladende Funktion entsprechend als friend-Funktion deklariert sein.
- Eine Funktion oder Klasse kann friend *mehrerer* (anderer) Klassen sein.
- Bei abgeleiteten Klassen vererbt sich die friend-Eigenschaft *nicht*.

Beispiel friend Aufruf

- Fkt() ist "normal" definiert (nicht Demo::Fkt()
- Funktion kann nur über ein Objekt (d.h. eine Instanz) auf Klassenvariablen zugreifen
  - Dann aber auch auf "private"
- Die Funktion muss auch "normal" aufgerufen werden.
  - Sie ist KEINE Member -Funktion

```
class Demo
{
      friend void Fkt(void);
    private:
      int iPriv;
    public:
      int iPub ;
    };

.....
void Fkt(void)
{
    iPub = 1;    //Fehler:
    Demo obj1;
    obj1.iPub = 2;
    obj1.iPriv = 3;
}
.....
Demo obj2;
obj2.iPub = 2;
obj2.iPriv = 3; //Fehler:
Fkt( );
obj2.Fkt( );    //Fehler:
```