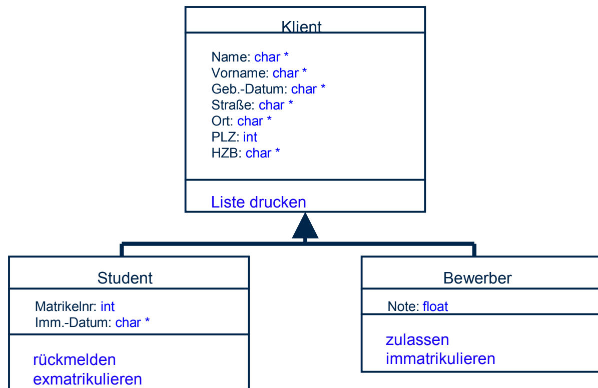
C++ - Vererbung

Übersicht

- Bisherige Verwendung von Klassen
- Vererbung
  - Grundlagen
  - Zugriffschutz
  - Vererbte Funktionen
- Zeiger auf Objekte (abgeleiteter) Klassen
- Virtuelle Funktionen
- Konstruktoren/Destruktoren

Bisherige Verwendung

- § Generierung eines Objekts und dessen Verwendung im Anwendungsprogramm (Instanzierung),
- § Verwendung eines Objekts zur Definition einer weiteren Klasse (Komposition).



```
class Point   {
   private:
     double xi, yi;
   public:
     void SetPoint(double x,
                   double y)
          {xi = x; yi = y;}
   };
class Circle  {
   private:
     double radius;
   public:
     Point center;
     void SetRadius(double r)
          {radius = r;}
   };
   .......
Circle objCircle;
objCircle.SetRadius(10.);
objCircle.center.SetPoint(1., 2.);
```

Ableitung und Vererbung von Klassen

In Erweiterung der bisherigen Funktionalität besteht die Möglichkeit zur Nutzung einer Klasse als Basis für die Definition einer *neuen* Klasse *ohne* auf Objekte der Basisklasse zurückgreifen zu müssen (Ableitung).

Eigenschaften:

- Die abgeleitete Klasse erbt alle Eigenschaften der Basisklasse, soweit sie in der abgeleiteten Klasse nicht *explizit* als unterschiedlich deklariert werden.
- Die abgeleitete Klasse kann *zusätzlich* weitere Daten und Funktionen definieren.
- Die abgeleitete Klasse kann Daten und Funktionen der Basisklasse *redefinieren*.
- Objekte der abgeleiteten Klasse sowie deren Member-Funktionen können (mit Einschränkungen) direkt auf die Member der Basis-Klasse zugreifen, als ob es ihre eigenen Member wären.
- Die abgeleitete Klasse kann wiederum zur Basisklasse für weitere Klassen werden. Damit entsteht eine Klassenhierarchie.
- Eine Ableitung kann im Regelfall auch dann durchgeführt werden, wenn die Basisklasse *nicht* im Source-Code vorliegt.

Syntax:

```
class abgeleiteterKlassenname : modifier basisKlassenname
```

Zugriffschutz beim Zugriff auf Klassenmenber

- Bei der Ableitung von Klassen erfolgt eine detaillierte Differenzierung, inwieweit auf
  - Datenelemente und Elementfunktionen der Basisklasse sowie auf
  - Datenelemente und Elementfunktionen der abgeleiteten Klasse

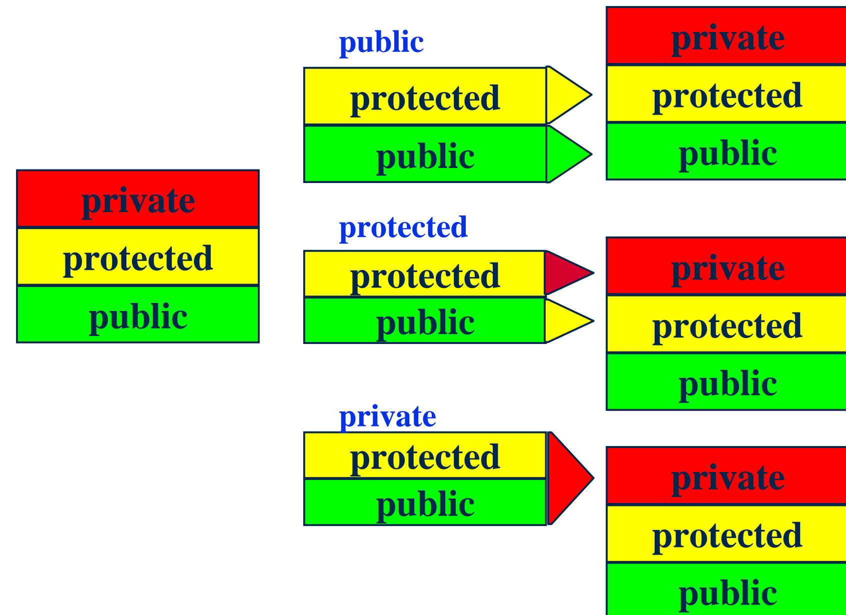
  zugegriffen werden kann.

- Ein Zugriff kann erfolgen durch:
  - Objekte der Basis & abgeleit. Klasse
  - Elementfunktionen (beider Klassen)
  - friend-Funktionen (beider Klassen)

```
class Point   {
   private:
     double xi, yi;
   public:
     void SetPoint(double x,
                   double y)
          {xi = x; yi = y;}
   };

class Circle  : public Point

{
   private:
     double radius;
   public:
     void SetRadius(double r)
          {radius = r;}

};

.......

Circle objCircle;

objCircle.SetRadius(10.);

objCircle.SetPoint(1., 2.);
```

Zugriffsschutz beim Zugriff auf Klassenmember durch Objekte

- § public
  - ø Der Zugriffsschutz für Elemente der Basisklasse bleibt unverändert.
    - o Objekte der abgeleiten Klasse haben grundsätzlich nur Zugriff auf public-Elemente der Basisklasse.
- § protected
  - ø public-Elemente der Basisklasse mutieren zu protected-Elementen,
  - ø protected-Elemente der Basisklasse mutieren zu private-Elementen
  - ø private-Elemente der Basisklasse bleiben unverändert.
- § private
  - ø Alle Elemente der Basisklasse mutieren zu private-Elementen.




```
class B {
protected:
  int a;
public:
  int b;
}

B obj;
onj.b = 0;
```

```
class D :
  public B {};
```

```
class D :
  protected B {};
```

```
class D :
  private B {};
```

```
D obj;
obj.a;
```

Zugriffsschutz beim Zugriff auf Klassenmember durch Elementfunktionen

- Elementfunktionen der Basisklasse
  - haben Zugriff auf alle Elemente der Basis-klasse.
  - haben keinen Zugriff auf irgendwelche Elemente einer abgeleiteten Klasse.
- Elementfunktionen der abgeleiteten Klasse
  - haben Zugriff auf alle Elemente der abgeleiteten Klasse
  - haben Zugriff auf die public- und protected-Elemente der Basisklasse unabhängig vom Modifier.
  - Achtung: Der Zugriff auf die erlaubten Elemente der Basisklasse erfolgt mit der gleichen Syntax wie der Zugriff auf die Elemente der abgeleiteten Klasse.

```
class Point   {
    private:
      double xi, yi;
    protected:
      int color;
    public:
      void SetPoint(double x,
                    double y)
           {xi = x; yi = y;}
    };
class Circle  : private Point
{
    private:
      double radius;
    public:
      void ResetPoint
           (int c, int x, int y)
           {color = c;
            SetPoint(x,y);}
      void SetRadius(double r)
           {radius = r;}
};
.......
Circle objCircle;
objCircle.SetRadius(10.);
objCircle.ResetPoint(10,1,2);
```

Vererbte Funktionen

- Eine Elementfunktion der Basisklasse, die auf eine abgeleitete Klasse vererbt wird, kann überschrieben werden, indem sie mit gleichem Namen und gleicher Signatur in der abgeleiteten Klasse neu definiert wird. Eine Überladung der Funktionen findet nicht statt.
- Wird die neue Elementfunktion über ein Objekt der abgeleiteten Klasse aufgerufen, so wird immer die Funktion der abgeleiteten Klasse benutzt. Die abgeleitete Klasse selbst kann die Basisklassenfunktion aber nach wie vor benutzen, indem sie den Bereichsoperator zur Kennzeichnung verwendet.
- Syntax:
  objektname.basisKlassenname::basisElementfunktionsname
  objektzeiger->basisKlassenname::basisElementfunktionsname

Empfehlung:

Verwenden Sie bei der Neudefinition von Elementfunktionen abgeleiteter Klassen nach Möglichkeit die ***vorhandene*** Funktionalität der Basisklasse durch direkten Aufruf.
Fügen Sie neue Funktionalität der ***Basisklasse*** hinzu, wenn erkennbar ist, dass auch andere, daraus abgeleitete Klassen diese verwenden können.

Vererbte Funktionen

Beispiel:

```
class Basis                          //Basisklasse
{
public:
   void Funktion(void);
   void Funktion(int i);             //überladene Funktion
};

class Sub  :  public Basis           //abgeleitete Klasse
{
public:
   void Funktion(int i);            //überschrieben Funktion
 };
.......
Sub objSub;                     //instanziiere Objekt
objSub.Funktion(10);            //nimm Funktion der abgeleiteten Klasse
objSub.Basis::Funktion( ); //nimm Funktion der Basisklasse
objSub.Basis::Funktion(20);//nimm überladene Funktion der Basisklasse
objSub.Funktion( );             //Fehler:Keine Überladung von Funktionen
                               //unterschiedlicher Bezugsbereiche
```

Zeiger auf Objekte/Klassen/…

- auf Instanz (Objekt)
  - Zugriff immer über Objekt oder Zeiger auf Objekt mittels "." oder "->"
  - Zugriffsschutz wird beachtet

- Subklasse kann auf
  - eigene und
  - geerbte (public) Funktionen/Variablen

  zugreifen

```
class Demo {
public:
    int var;
    int func ();
}
class Sub : public Demo {}

// Zeiger auf Objekte
Demo obj;                // Stack
Demo *pObj1 = &obj;      // Zuweisung
Demo *pObj2 = new Demo;// Heap

obj.var = 10;
cout << pObj1->var;      // obj.var
pObj2->func();   // Aufruf func

Sub  sobj;              // Stack
Sub *psObj = &sobj;     // Zuweisung
sobj.var = 20;          // public Var
psObj->func();         // public func
```

Zeiger auf Objekte abgeleiteter Klassen

§ Ein Zeiger auf ein Objekt der abgeleiteten Klasse kann die Adresse auf ein Objekt der Basisklasse nur nach expliziter Typkonvertierung (cast) aufnehmen:

```
Basis objBasis;
Basis *ptrBasis = &objBasis;
Sub *ptrSub = (Basis *)ptrBasis;
```

§ Ein Zeiger auf ein Objekt der Basisklasse kann die Adresse auf ein Objekt der abgeleiteten Klasse ohne explizite Typkonvertierung aufnehmen:

```
Sub objSub;
Basis *ptrBasis = &objSub;
```

Zeiger auf Objekte abgeleiteter Klassen

- § Eigenschaften eines Zeigers auf ein Objekt der Basisklasse, dem die Adresse auf ein Objekt der abgeleiteten Klasse zugewiesen wurde:
  - ø Der Zeiger kann (soweit es der Zugriffsschutz erlaubt) auf Elemente der abgeleiteten Klasse zugreifen, die diese von der Basisklasse geerbt hat,
  - ø Der Zugriff auf die nicht vererbten Elemente der abgeleiteten Klasse ist nur mit einer expliziten Typkonvertierung (cast) des Zeigers möglich.
  - ø Der Zeiger kann auf Elementfunktionen zugreifen, die in der Basisklasse und in der abgeleiteten Klasse den gleichen Funktionsnamen aufweisen. Erfolgt der Zugriff ohne Typkonvertierung, so wird immer die vererbte Funktion aus der Basisklasse aufgerufen. Für den Zugriff auf die nicht vererbte Funktion ist eine explizite Typkonvertierung (cast) erforderlich.

- § (Wird die Elementfunktion als virtuelle Funktion definiert und der Zugriff erfolgt ohne Typkonvertierung, so wird immer die nicht vererbte Funktion aus der abgeleiteten Klasse aufgerufen)

Zeiger auf Objekte abgeleiteter Klassen

```
class Basis                              //Basisklasse
{
public:
    int BasisVar;
    void BasisFunktion(void);
    void Funktion(void);
};

class Sub   :  public Basis    //abgeleitete Klasse
{
public:
    int SubVar;
    void SubFunktion(void);
    void Funktion(void);
};

Sub objSub;                          //instanziiere Objekt der abgeleiteten Klasse
Basis *ptrBasis = &objSub;           //Definition und Initialisierung
ptrBasis->BasisVar = 10;            //Aktion auf vererbte Variable
((Sub *)ptrBasis)->SubVar = 20;//Aktion auf nichtvererbte Variable
ptrBasis->BasisFunktion( );        //Aufruf vererbte Funktion
((Sub *)ptrBasis)->SubFunktion( ); //Aufruf nichtvererbte Funktion
ptrBasis->Funktion( );               //Aufruf vererbte Funktion
((Sub *)ptrBasis)->Funktion( );    //Aufruf nichtvererbte Funktion
```

Polymorphie & Virtuelle Funktionen

- Problemstellung:
  - Man möchte aus einer Liste von Zeigern auf Objekte auf die Basisklasse eine Funktion dieser Objekte aufrufen. Dabei soll, falls vorhanden, die überladenen Funktion der SubKlasse und nicht die Funktion der Basisklasse aufgerufen werden.
- Polymorphy
  - D.h. es soll automatisch immer die Funktion/Methode „passend“ zur Klasse aufgerufen werden, auch wenn man nur einen Zeiger auf die Basisklasse besitzt.

Polymorphie & Virtuelle Funktionen

§ Welche Methoden stehen uns zu Verfügung, um kontextspezifisch alternative Funktionen zur Ausführung zu bringen:

1. Man verwende eine Reihe von switch-Anweisungen, gesteuert von einem anwenderselektiven Index.
   Problem: Jede neue Option führt zu umfangreicher Modifikation und Neukompilierung des Programms.
2. Man verwende einen Zeiger auf Funktionen, dessen aktueller Inhalt die aufzurufende Funktion selektiert.
   Problem: Programmmodifikationen sind zwar geringer als bei Methode (1) aber noch immer vorhanden.
3. Man verwende einen Zeiger auf eine Objekt einer Basisklasse und initialisiere ihn mit der Adresse des Objekts einer abgeleiteten Klasse.
   Problem: Die Verwendung des Zeigers zum Aufruf der Elementfunktion ruft die vererbte Funktion der Basisklasse auf.
4. Man verwende Methode (3) und definiere die Elementfunktion als virtuelle Funktion.
   Folge: Die Verwendung des Zeigers zum Aufruf der Elementfunktion ruft die Funktion der abgeleiteten Klasse auf.

Virtuelle Funktionen

Beispiel:

```
class Basis                                    //Basisklasse
{
public:
   virtual void Funktion(void);
};

class Sub  :  public Basis                     //abgeleitete Klasse
{
public:
   virtual void Funktion(void);
};

void FunSelektor(const Basis &obj)
{
   obj.Funktion;    //Aufruf der Elementfunktion der abg. Klasse

}

Sub objSub;          //instanziiere Objekt der abgeleiteten Klasse
FunSelektor(objSub); //Zeiger auf objSub als Aktualparameter
```

Virtuelle Funktionen - Eigenschaften

- Virtuelle Funktionen müssen Elementfunktionen von Klassen sein.
- Das Schlüsselwort virtual muss bei der Deklaration der Elementfunktion stehen.
- Alle Elementfunktionen abgeleiteter Klassen, die einen identischen Prototyp aufweisen sind automatisch virtuell. Das Schlüsselwort virtual muss in der abgeleiteten Klasse nicht wiederholt werden.

  Empfehlung: Wiederholen Sie das Schlüsselwort in der abgeleiteten Klasse.
  Ist der Prototyp unterschiedlich, dann ist die Elementfunktion in der abgeleiteten Klasse nicht mehr virtuell, auch wenn das Schlüsselwort virtual dort wiederholt wird.
- Ist eine virtuelle Funktion Elementfunktion einer Basisklasse, so muss sie neben der Deklaration dort auch definiert werden.
  Ausnahme: rein virtuelle Funktionen: virtual rückgabetyp name(parameter) = 0;
- Die Definition virtueller Funktionen ist identisch zur Definition normaler Elementfunktionen. Das Schlüsselwort virtual muss nicht wiederholt werden.
- Virtuelle Funktionen können auch als inline-Funktion definiert werden.
- Virtuelle Funktionen können nicht als statische Funktion definiert werden.

Virtuelle Funktionen - Beispiel

```
class Basis                             //Basisklasse
{
public:
    void Funktion1(int i);      //keine virtuelle Methode der Basis-Klasse
    virtual int Funktion2(void);//virtuelle Methode der Basis-Klasse
    virtual double Funktion3(int j);//virtuelle Methode der Basis-Klasse
};

class Sub   :  public Basis             //abgeleitete Klasse aus Basis
{
public:
    virtual void Funktion1(int i); //keine virtuelle Methode der
                                    //abgeleiten Klasse
    virtual int Funktion2(void);   //virtuelle Methode der abgel. Klasse
    virtual double Funktion3(void); //keine virt. Meth. der abgel. Klasse
};

class SubSub  :  public Sub             //abgeleitete Klasse aus Sub
{
public:
    virtual double Funktion3(int j); //virtuelle Methode der abgel.Klasse
};
```

Virtuelle Funktionen – weitere Eigenschaften

- Wird eine virtuelle Funktion der Basisklasse nicht in der abgeleiteten Klasse redeklariert und redefiniert, so nutzen die Objekte der abgeleiteten Klasse die Elementfunktionen der Basisklasse.
- Wird eine virtuelle Funktion in der abgeleiteten Klasse redeklariert, so muss sie neben der Deklaration dort auch definiert werden.
- Virtuelle Funktionen, die in einer abgeleiteten Klasse redefiniert wurden, können nach wie vor auf die virtuellen Funktionen der Basisklasse mittels des Klassenbezugsoperators zugreifen.

```
•class Basis
{
public:
  virtual int Funktion(int i)
              {return i;}
};

class Sub  :  public Basis
{
public:
  virtual int Funktion(int j)
      {return 2*j +
              Basis::Funktion(j);}
};
```