Objektorientierte Programmierung (OOP)

Einführung in die OOP

Die objektorientierte Programmiertechnik wurde in den 80-er Jahren entwickelt und war der Ausweg aus der Software Krise. Die umfangreichen Programme mit grafischen Benutzeroberflächen waren kaum mehr mit prozeduralen Methoden beherrschbar.
Die OOP mit der Verwendung von Klassenbibliotheken gehört heute zu den Standard Werkzeugen zur Erstellung von GUI-Programmen. (Visual C++).

Die moderne **Software Entwicklung** erfordert Methoden, mit denen die Überschaubarkeit, die Wiederverwendbarkeit und die Erweiterbarkeit besser unterstützt wird als mit der prozeduralen Technik.

Die **OOP** erfüllt dazu folgende Punkte :

- syntaktische Einheiten für Daten und Funktionen → Objekte
- schmale Schnittstellen (minimale Anzahl von Parametern) → Daten in Objekten
- wenige Schnittstellen → Methoden in Objekten
- explizite (erkennbare) Schnittstellen für Datenaustausch → öffentliche Daten
- geschützte Daten im Modul → private Datenelemente
- zur Wiederverwendbarkeit - Gemeinsamkeiten nutzen → Vererbung
- zur Erweiterbarkeit - Darstellungsunabhängigkeit → Polymorphie

OOP in C++

Die objektorientierte Programmierung in C wurde mit C++ im Jahr 1984 von Bjarne Stroustrup eingeführt. C++ enthält dabei C als Untermenge und wird C immer mehr ablösen.
C++ ist eine der meist verwendeten Sprachen und wird auf vielen Plattformen, wie UNIX, MS-Windows eingesetzt.
Neue Sprachentwicklungen (Java) lehnen sich auch sehr stark an die Sprache C++.

Grundlegende Begriffe : Klasse, Objekt und Elemente
Eine **Klasse** ist ein Datentyp, der Daten und Funktionen (Methoden) enthält.
Ein **Objekt** (Instanz) ist eine konkrete Ausführung einer Klasse, entspricht einer Variablen.
Die **Elemente** (Eigenschaften) einer Klasse sind Daten und Funktionen (Methoden)

C++ unterstützt folgende objektorientierte Ansätze :

- **Datenabstraktion** (data abstraction) =
  Trennung von Definition (Klasse) und Objekt (Instanz).
- **Kapselung** (encapsulation) =
  der Zugriff auf die Elemente einer Klasse kann nach außen geschützt werden.
- **Vererbung** (inheritance) =
  die Eigenschaften (Elemente) einer Klasse (Basisklasse) werden an eine neue Klasse (abgeleitete Klasse) weitergegeben (vererbt).
- **Polymorphie** (polymorphism) =
  Polymorphie ist der Begriff für Vielgestaltigkeit.
  In der OOP ist darunter die Fähigkeit zu verstehen, Methoden von Basisklassen in abgeleiteten Klassen neu zu definieren und damit in neuer Gestalt auszuführen.

Datenabstraktion :

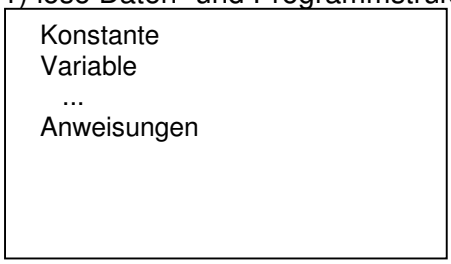
Die Verwendung von Objekten setzt die Typdefinition einer Klasse voraus.
Eine Klasse ist ein abstrakter Datentyp, der Daten und Funktionen enthalten kann.
Die Variable zu einer Klasse ist ein Objekt (Instanz) und belegt Speicherplatz.

Kapselung :

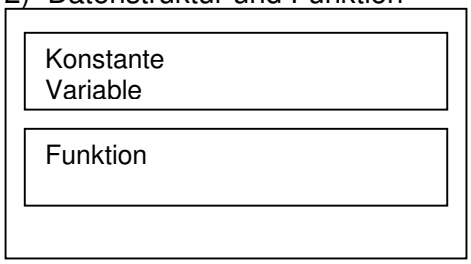
Die Elemente einer Klasse sind zu einer Einheit verbunden.
Der **Zugriff** auf die Elemente einer Klasse kann mit den Zugriffsspezifizierern `private`,`public` und `protected` gesteuert werden.
Private Elemente sind geschützt und nur über Methoden der eigenen Klasse zugreifbar.
Public Elemente sind öffentlich und auch von außen zugreifbar.
Protected Elemente sind geschützt, jedoch auch in abgeleiteten Klassen zugreifbar.
Über Freundschaften (**friends**) kann der Zugriffsschutz auf private Elemente aufgehoben werden.

Schema der Datenkapselung der OOP und Vergleich zur prozeduralen Programmierung :

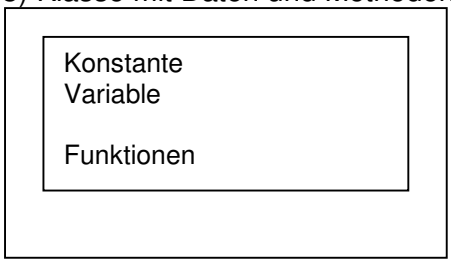
1) lose Daten- und Programmstruktur



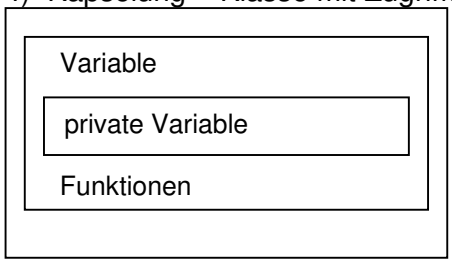
2) Datenstruktur und Funktion



3) Klasse mit Daten und Methoden



4) Kapselung = Klasse mit Zugriffsschutz



Vererbung :

Neue Klassen können von vorhandenen Klassen (**Basisklassen**) abgeleitet werden.
Abgeleitete Klassen erben damit die Eigenschaften der Basisklassen.
Eine abgeleitete Klasse kann zu den geerbten Eigenschaften zusätzlich neue eigene Eigenschaften erhalten.
Man unterscheidet die einfache und mehrfache Vererbung. Bei einfacher Vererbung wird nur von einer Basisklasse, bei mehrfacher Vererbung von mehreren Basisklassen geerbt.
C++ unterstützt als eine der wenigen Sprachen auch die Mehrfachvererbung.

Polymorphie :

Polymorphie bedeutet die Fähigkeit, Methoden der Basisklasse in abgeleiteten Klassen neu zu definieren (redefinieren, überschreiben), um damit in abgeleiteten Klassen neue Aufgaben unter gleichem Funktionsnamen auszuführen. Damit der Aufruf von überschriebenen Methoden problemlos funktioniert sind in C++ virtuelle Methoden zu definieren.
Virtuelle Methoden werden erst zur Laufzeit über eine virtuelle Methodentabelle (virtual method table) ausgewählt. Diesen Vorgang nennt man spätes Binden (late binding) im Gegensatz zum frühen Binden (early binding) beim Übersetzungsvorgang.

Klasse :

Eine Klasse ist ein benutzerdefinierter Datentyp, der mit **class** definiert wird.

Klassendefinition :

```
class < KlassenName >
{ < Datenelemente >
  < Methoden      >
};
```

Beispiel : die Klasse Person

```
class  Person                  ////////  Klasse Person  ////////
{                              //  Datenelemente
   public:                     //  öffentliche Datenelemente :
      char name[20];           //     Name als String
   protected:                  //  geschützte  Datenelemente :
      int alter;               //     Alter als Integer
   private :                   //  private     Datenelemente :
      char beruf[40];          //     Beruf als String
                               //  Methoden (Elementfunktionen)
   public:                     //  öffentliche Methoden :
      void eingabe();          //      zur Daten-Eingabe
      void ausgabe();          //      zur Daten-Ausgabe

};                             ////  Ende der Klasse Person ////
```

Die **Klasse** Person enthält verschiedenene Datenelemente (wie eine Struktur), um die notwendigen Daten aufzunehmen, aber zusätzlich auch Methoden, um die notwendigen Bearbeitungen dieser Daten durchzuführen.
Die **Zugriffskontrolle** erfolgt mit Zugriffsspezifizierer, wie **public, protected und private.**
Private und protected Elemente können nur über Methoden der Klasse verabeitet werden, public Elemente sind auch von außen (wie die Komponenten von Strukturen) zugreifbar.
Die Voreinstellung (ohne Spezifizierer) ist bei Klassen private, bei Strukturen public.
Die Methoden werden im allgemeinen nur als Funktionsprototypen in der Klasse angegeben und die **externe Methodendefinitionen** dann außerhalb der Klasse ausgeführt.
Eine komplette Funktionsdefinition innerhalb der Klasse wird als Inline Funktion ausgeführt und hat die Wirkung, dass bei der Compilation der Aufruf durch den Funktionskörper ersetzt wird.
(vorteilhaft bei kurzen, rasch auszuführenden Funktionen)

externe Methodendefinition :

```
Typ <KlassenName>::<MethodenName>(Parameterliste)
{  < Funktionsblock>
}
```

Die Angabe der Klasse mit **Bereichsoperator "::"** vor dem Methodennamen ist notwendig und ermöglicht den Zugriff auf die Elemente der Klasse.

Beispiel : externe Methodendefinition zur Klasse Person

```
void Person::eingabe()         //////  Definition eingabe() //////
{  cout << " Name    : " ; cin >> name;         // Zugriff auf public El.
   cout << " Alter   : " ; cin >> alter;        // Zugriff auf protected El.
   cout << " Beruf   : " ; cin >> beruf;        // Zugriff auf private El.
}
```

Die Methode void eingabe() ist innerhalb der Klasse Person deklariert (durch den Prototyp) und wird außerhalb der Klasse definiert (externe Methodendefinition).

Objekt (Instanz) :

Ein Objekt (auch Instanz genannt) ist eine Variable zu einer Klasse.
Beim Instanzieren wird Speicherplatz für ein Objekt bereitgestellt.

Objektdefinition : `KlassenName  ObjektName ;`

Zugriff auf Datenelemente und Methoden : nur für public - Elemente möglich !

```
ObjektName.Datenelement ;
ObjektName.Methode();
```

Beispiel : Instanz zur Klasse Person

```
Person mann;            //  Instanz mann zur Klasse Person

mann.name;              //  Zugriff auf ein Datenelement der Klasse Person
mann.eingabe();         //  Aufruf einer Methode der Klasse Person
```

Konstruktor :

Ein Konstruktor ist eine **spezielle Methode** der Klasse **zur Initialisierung** einer Instanz.
Der Konstruktor ist dadurch gekennzeichnet, dass er denselben Namen wie die Klasse hat.
Wird kein expliziter Konstruktor definiert, so wird automatisch ein Standardkonstruktor ausgeführt.

Konstruktordefinition :

```
<KlassenName>::<KlassenName>(Parameterliste)
{  < Funktionsblock mit Wertzuweisungen >
}
```

Ein Konstruktor hat **keinen Return Typ !** (auch nicht void)
Für eine Klasse können auch mehrere, überladene Konstruktoren erstellt werden,
um damit Initialisierungen auf unterschiedliche Arten zu ermöglichen.
Konstruktoren können auch vorteilhaft als inline Funktionen ausgeführt werden.

Beispiel : Konstruktor zur Klasse Person

```
class  Person                           ////////  Klasse Person  ////////
{                                       //  Datenelemente
   public:                              //  öffentliche Datenelemente :
        char name[20];                  //     Name
          ...
   public:                              //  öffentliche Methoden :
        void eingabe();                 //      zur Daten-Eingabe
         ...

        Person(char *nam,int alt,char *ber); // Konstruktor zur Klasse Person

        Person(char *nam)               //  2.(überladener) Konstruktor
        { strcpy(name,nam);             //  als Inline Definition
        }
         ...
};                                      ////  Ende der Klasse Person ////

Person::Person(char *nam,int alt,char *ber) // externe Konstruktor Definition
{  strcpy(name,nam);                    //  zur Initialisierung
   alter=alt;                           //  von Datenelementen
   strcpy(beruf,ber);
}
                                        //  Instanzieren über Konstruktor-Aufruf
Person sohn("Peter",18,"Schüler");      //  Instanz sohn über 1.Konstruktor
Person frau("Fr.Hübsch");               //  Instanz frau über 2.Konstruktor
Person klone = sohn;                    //  Initialisierung über Kopierkonstruktor
```

Inline Elementfunktionen :

Inline Funktionen sind vorteilhaft bei kurzen, rasch auszuführenden Funktionen.
Inline Funktionen werden durch den Compiler nicht in Funktionsaufrufe übersetzt, sondern werden an allen Stellen der Aufrufe eingefügt.
Es gibt 2 Ausführungsmöglichkeiten :

- Deklaration und Definition innerhalb der Klasse

```
class  Person
{  ...
   void ausgabe()                    // Inline Definition
   {  printf(" Name    : %s",name);
   }
};
```

- Deklaration in der Klasse und Definition außerhalb (übersichtlicher)

```
class  Person
{  ...
   void ausgabe();                   //  Deklaration
   ...
};

inline void Person::eingabe()        //  Inline Definition
außerhalb der Klasse
{  printf(" Name    : " ); scanf("%s",&name);
}
```

Überladene Funktionen :

Überladene Funktionen sind Funktionen mit gleichen Funktionsnamen, jedoch unterschiedlicher Signatur (unterschiedliche Anzahl oder Typen der Parameter).
Der Compiler kann an der Aufrufstelle die richtige Funktion aus dem Vergleich der Typen der aktuellen mit den formalen Parameter herausfinden und aufrufen.
Überladene Funktionen können z.B. für gleichartige Operationen verwendet werden, die mit unterschiedlichen Datentypen erledigt werden sollen.

```
float quadrat ( float x )                // Funktion quadrat für float-Werte
{  return x*x;
}

long quadrat( long x )                   // 2.Funktion quadrat für long-Werte
{  return x*x;
}
```

Kopier - Konstruktor :

Der Kopier-Konstruktor dient **zur Initialisierung** mit einem Objekt (der eigenen Klasse).
Es wird automatisch ein Default-Copy-Konstruktor ausgeführt, welcher auf Bitebene (flache Kopie) alle Elemente eines Objekts auf die neue Instanz kopiert.
Ein expliziter Copy-Konstruktor ist nur dann notwendig, wenn dynamische Datenelemente verwendet werden und eine tiefe Kopie (Inhalt von Speicheradressen) erforderlich ist.

Beispiel : Verwendung des Default-Kopier-Konstruktors

```
Person mann;            //  Objekt mann der Klasse Person
Person erbe = mann;     //  Default Copy-Konstruktor
                        //  kopiert alle Datenelemente von mann auf erbe
```

Definition eines expliziten Kopier-Konstruktors :

```
           <KlassenName>( KlassenName& x)
           {  < Funktionsblock mit Wertzuweisungen >
           }
```

Statische Klassenelemente (static) :

Jede Instanz einer Klasse legt die Datenelemente an eigenen Speicherplätzen an.
Ein **statisches Datenelement** hingegen wird nur einmal für alle Instanzen einer Klasse erzeugt und damit wird von allen Instanzen der Klasse auf das gleiche Datenelement zugegriffen.
Dies könnte auch mit globalen Variablen erreicht werden, widerspricht jedoch den Grundsätzen der objektorientierten Programmierung.

```
class Teilnehmer
{    static int anzahl;                    //  statisches Datenelement

     teilnehmer(int nr)                    //  Konstruktor
     {  nummer=nr; anzahl++;
     }
}

int teilnehmer::anzahl=0;                  //  static Element initialisieren !
```

Konstante Datenelemente :

Konstante können im Klassenblock nicht direkt initialisiert werden, ausgenommen sie werden mit static definiert.

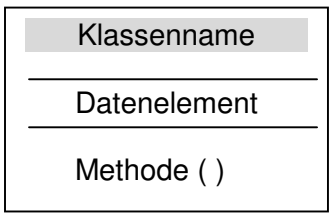
Beispiel :

```
class Konstante1
{    static const int N=10;          //  static Konstante
}

class Konstante2
{    const float E;                  //  Konstante
     Konstante2():E(2.718)           //  mit Konstruktor Initialisierungsliste
     {  }
}
class Konstante3
{    enum { K=20 };                  //  int-Konstante über enum
}
```

Klassen Diagramme :

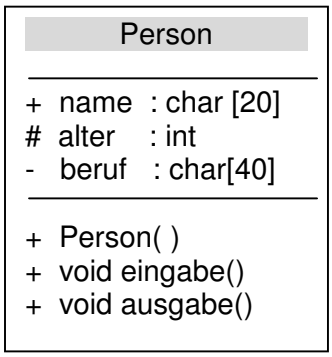
Klassendiagramme stellen Klassen und Klassenbeziehungen in Diagrammform dar.

Klassendiagramm :



Zugriffsrechte
+ public
# protected
- private

Beispiel Person :



Vererbung :

Vererbung ist ein wichtiges Konzept zur Wiederverwendbarkeit von Programmteilen.
Abgeleitete Klassen erben die Eigenschaften von Basisklassen und haben zusätzlich eigene Eigenschaften.

Vererbung :

```
class <KlassenName> : <Spezifizierer> <Basisklasse>
{ < Datenelemente >
  < Methoden      >
};
```

Vererbung von Zugriffs-Rechten :

Spezifizierer in Basisklasse	Spezifizierer bei Ableitung	Zugriff in abgeleiteter Klasse
public	public	public
protected	public	protected
private	public	Nicht möglich
public	protected	protected
protected	protected	protected
private	protected	Nicht möglich
public	private	private
protected	private	private
private	private	Nicht möglich

Private-Elemente sind in abgeleiteten Klassen nicht zugreifbar.
Bei den anderen Zugriffsrechten begrenzt der Ableitungs-Spezifizierer die maximalen Zugriffsmöglichkeiten in den abgeleiteten Klassen auf seinen Wert.
Die Voreinstellung des Ableitungs-Spezifizierer ist bei Klassen privat, bei Strukturen public.

Beispiel : die abgeleitete Klasse Kunde

```
class  Kunde : public Person  ///////  Klasse  Kunde  ////////
                              //  abgeleitet von Klasse Person
                              //  mit Zugriffspez. public
{  public:
      int kundennummer;       //  Kunden Nummer
   private :
      float kontostand;       //  Konto Stand
   public:
      void konto_eingabe();   //  Eingabe der Daten
      void konto_abfrage();   //  Ausgabe der Daten
      Kunde(char *nam,int alt,char *ber,int knr); //  Konstruktor
};                            ////  Ende der Klasse Kunde  ////
```

Konstruktor bei abgeleitenden Klassen :

Der Konstruktor einer abgeleiteten Klassen kann Parameter als Argumente an den Basisklassenkonstruktor weitergeben und eigene Datenelemente initialisieren.

Konstruktordefinition :

```
<KlassenName>::<KlassenName>(Parameterliste):<Basisklasse>(Argumente)
{  < Funktionsblock mit Wertzuweisungen >
}
```

Beispiel : Konstruktor der abgeleiteten Klasse Kunde

```
Kunde::Kunde(char *nam,int alt,char *ber,int knr): Person(nam,alt,ber)
                              //  Konstruktor
                              //  kopiert Werte an den Basis-Konstruktor
                              //  und
{  kundennummer=knr;          //  initialisiert eigene Datenelemente
}
```

Beispiel : Zugriff auf Eigenschaften der Basisklasse

```
void Kunde::konto_abfrage()
{  cout << "\n Nr    : " << kundennummer;   // eigenes Element
   cout << "\n Name  : " << name;           // Element der Basisklasse
   cout << "\n Alter : " << alter;          // protected El. der Basisklasse
   cout << "\n Konto : " << kontostand;     // eigenes private Element
}
//-------------------------------------------------------------------------
//                        Hauptprogramm
void main()
{ int i;
  Kunde  neu("Stroustrup",55,"Autor",1234);// Konstruktor Aufruf
   ...
  neu.ausgabe();                           // Methode der Basisklasse
  strcpy(neu.name,"Bjarne Stroustrup");    // public Element der Basisklasse
  neu.konto_abfrage();                     // Methode der eigenen KLasse
   ...
}
```

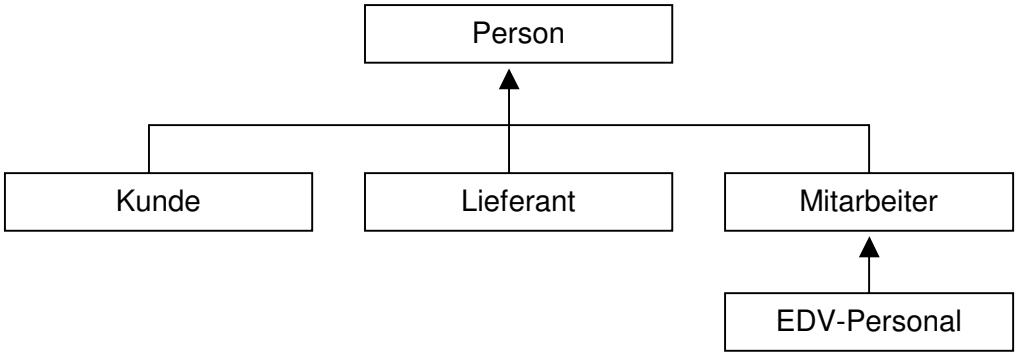
Mehrfach-Vererbung :

Eine abgeleitete Klasse erbt von mehreren Basisklassen.

```
class <KlassenName> : <Spezifizierer> <Basisklasse1>,
                      <Spezifizierer> <Basisklasse2>,
                      <Spezifizierer> <Basisklasse3>
{ < Datenelemente >
  < Methoden      >
};
```

Klassen-Hierarchie :

Über Vererbung kann eine hierarchische Beziehung zwischen Klassen aufgebaut werden. Beim Design ist darauf zu achten, dass Basisklassen generell und die abgeleiteten Klassen immer spezieller auszuführen sind.



Freundschaften (friends) :

Über Freundschaften (friends) kann ein Zugriff auf geschützte (private) Elemente einer Klasse von außen erfolgen. Diese Art des Zugriffes auf geschützte Klassenelemente sollte jedoch nur für Ausnahmefälle herangezogen und im allgemeinen sollte der Zugriff durch Klassen- und Vererbungs- Design geregelt werden.
Über friends können fremde Klassen und Funktionen einen Zugriff auf nicht public- Elemente erhalten. Friend-Klassen und Friend-Funktion werden über den Spezifizierer friend vereinbart. Friends müssen in der Klasse, auf die sie zugreifen dürfen, mit friend ausgewiesen sein. Freundschaften werden nicht an abgeleitete Klassen vererbt.

friend Definition :

```
class <KlassenName>
{  ...
   friend <KlassenName>;
   friend <Funktion>;
   ...
};
```

Die Klasse `<KlassenName>` und die Funktion `<Funktion>` dürfen auf die geschützten Elemente der Klasse `<KlassenName>` zugreifen.

Beispiel : Friend-Klasse zur Klasse Person

```
class  Person
{   ...
  public:
    ...
    friend class P_Analyse;         //  Freundschaft mit der Klasse Analyse
    ...
};

class Analyse                       //  Basisklasse  Analyse
{ public:
    void  altersgruppe(int alter)
    {  if ((alter>0 )&&(alter<=20))  cout << "\n Junior ";
       if ((alter>20)&&(alter<=60))  cout << "\n Erwachsener ";
       if ((alter>60)&&(alter<=100)) cout << "\n Senior ";
    }
};

class P_Analyse : public Analyse    //  abgeleitete Klasse
{ public:
    void  altersgruppe(Person &p)   //  überladene Funktion
    {  Analyse::altersgruppe(p.alter);    //  Zugriff auf protected Element
    }                                     //  von Person
};

//////////////////          Hauptprogramm           //////////////////////
void main()
{ ...
  Person mann; ...
  P_Analyse a;                      //  Instanz zur Klasse P_Analyse
  ...
  mann.ausgabe();
  a.altersgruppe(mann);             //  Ausgabe der Altersklasse von mann
  ...
}
```

Polymorphie :

Polymorphie bedeutet die Fähigkeit, Methoden der Basisklasse in abgeleiteten Klassen neu zu definieren (auch überschreiben oder redefinieren bezeichnet), um damit in abgeleiteten Klassen neue Aufgaben unter gleichem Funktionsnamen auszuführen.
Überschreiben bedeutet, dass Methoden mit gleicher Signatur nochmals definiert werden. Gleiche Signatur heißt, dass Anzahl und Typen der Parameter übereinstimmen und der Rückgabewert entweder übereinstimmt oder von dem der Basismethode abgeleitet ist.
Überschreiben hat eine Ähnlichkeit zu überladenen Methoden, unterscheidet sich jedoch durch diese Anforderung der gleichen Signatur und ermöglicht, Methoden zwar einheitlich aufzurufen, jedoch in verschiedenen, abgeleiteten Klassen mit unterschiedlichen Definitionen auszuführen.
Damit das Überschreiben von Methoden problemlos funktioniert sind dazu in C++ virtuelle Methoden zu definieren.
Virtuelle Methoden werden mit dem Spezifizierer **`virtual`** deklariert und müssen in allen abgeleiteten Klassen die gleiche Signatur besitzen und ebenfalls virtual ausgeführt werden.
Über virtuelle Methoden wird ein dynamisches Binden (spätes Binden, late binding) ausgeführt.
Bei der späten Bindung wird erst zur Laufzeit die Adresse der virtuellen Methode aus einer Tabelle mit Referenzen (**virtual method table**) ausgewählt. Nur so kann erreicht werden, dass aus einer Klassenhierarchie die richtige ausgewählt wird.
Beim normalen frühen Binden wird bereits beim Übersetzungsvorgang die Adresse der Methode bei der Aufrufstelle festgelegt. Damit wird bei Referenzen und Zeiger auf Objekte nicht die gewünschte überschriebene Methode der abgeleiteten Klasse angesprochen.

Definition virtueller Methoden : `virtual <Typ><MethodenName>(Parameterliste);`

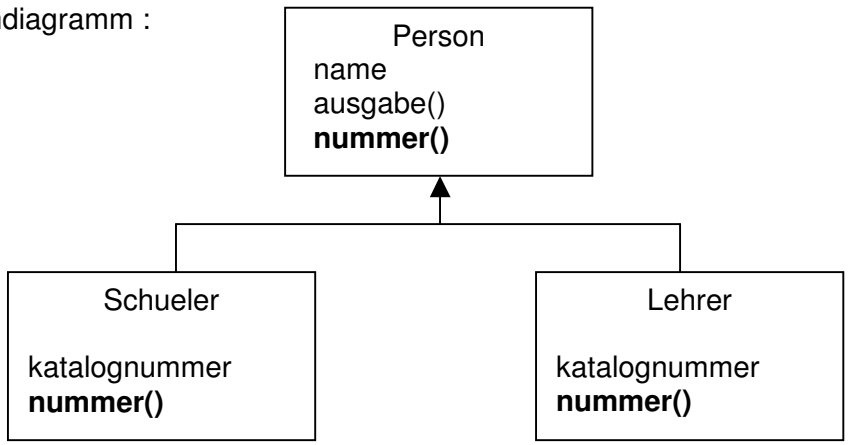
Beispiel :

```
////////////////////////////////////////////////////////////////////////
//              Klasse Person mit virtueller Methode nummer()
class  Person
{ public:
    char name[20];
      ...
  public:
    void ausgabe();                 // Ausgabe der Daten
    virtual void nummer()           // virtuelle Methode nummer()
    { return;}
      ...
};
////////////////////////////////////////////////////////////////////////
//                    externe Methoden Definitionen
void Person::ausgabe()
{  printf("\n Name    : %s",name  );
   nummer();                  // Aufruf der virtuellen Methode nummer()
}
////////////////////////////////////////////////////////////////////////
//              abgeleitete Klassen mit virtueller Methode nummer()
class  Schueler : public Person     //  abgeleitete Klasse Schueler
{ public:
    int katalognummer;
  public:
    virtual void nummer();          //  virtuelle Methode nummer()
};
class Lehrer : public Person        //  abgeleitete Klasse Lehrer
{ public:
    long personalnummer;
  public:
    virtual void nummer();          //  virtuelle Methode nummer()
};
```

```
////////////////////////////////////////////////////////////////////////
//                    externe Methoden Definitionen

void Schueler::nummer()             //  Ausgabe der Katalognummer
 {  cout << " Katalognummer = " << katalognummer;
 }

void Lehrer::nummer()               //  Ausgabe der Personalnummer
 {  cout << " Personalnummer = " << personalnummer;
 }


////////////////////////////////////////////////////////////////////////
//                            Hauptprogramm

 void main()
 {
   Schueler listig("Listig",15,"Schüler");
   Lehrer   specht("Specht",59,"Lehrer") ;

   listig.katalognummer = 10;
   specht.personalnummer = 1234567890;

   listig.ausgabe();                // Ausgabe der Katalognummer ->
                                    // nummer() von Schueler wird aufgerufen !
   specht.ausgabe();                // Ausgabe der Personalnummer ->
                                    // nummer() von Lehrer wird aufgerufen !
 }
```

Die **Methode nummer()** wird von der Methode ausgabe() der Basisklasse Person aufgerufen.
In den abgeleiteten Klassen Schueler und Lehrer wird die Methode nummer() überschrieben.
Die virtuelle Methode Schueler::nummer() dient zur Ausgabe der Katalognummer,
die virtuelle Methode Lehrer::nummer() hingegen zur Ausgabe der Personalnummer.
Die **virtuelle Methode nummer()** wird somit vielgestaltig eingesetzt.

Klassendiagramm :



Dynamische Objekte :

Dynamische Objekte können mit new erzeugt werden. Dazu muß ein Zeiger auf die Klasse vereinbart werden. Der Zugriff auf das Objekt erfolgt dann über den Zeiger. Die Freigabe des dynamischen Objektes wird mit delete ausgeführt.

Zeiger auf Klasse :

```
Klasse* Zeiger;
```

Dynamisches Anlegen eines Objektes :

```
Zeiger = new Klasse;
```

oder über Konstruktoraufruf :

```
Zeiger = new Klasse(Argumente);
```

Zugriff auf Elemente :

```
Zeiger->Datenelement;
Zeiger->Methode(Argumente);
```

Freigabe eines dynamischen Objektes :

```
delete Zeiger;
```

Beispiel :

```
Person *p,*liste;                    //  Zeiger auf Klasse Person
Kunde *k;                            //  Zeiger auf abgeleitete Klasse Kunde

p = new Person;                      //  dynamisches Objekt mit new
p = new Person("Konstrukteur",1,"");//  auch über Konstruktoraufruf

printf("\n Name : %s ",p->name);     //  Zugriff auf Datenelement
p->eingabe();                        //  Aufruf einer Methode

k = new Kunde;                       //  dyn. Objekt von Kunde
p = k;                               //  Zuweisung auf Zeiger der Basisklasse
k = (Kunde*)p;                       //  umgekehrt nur über type-casting
delete k;                            //  Freigabe mit delete

liste=new Person[5];                 //  dynamisches Feld von Objekten

for (i=0;i<5;i++)
{  printf("\n Name : %s ",liste[i].name);
}
delete[] liste;                      //  Freigabe des dynamischen Feldes
```

Zeiger this :

Das **Schlüsselwort this** stellt einen Zeiger auf das eigene Objekt einer Klasse dar.
Mit this kann daher innerhalb einer Klasse auf die eigene Instanz verwiesen werden.
Mit *this kann das ganze Objekt selbst (mit den aktuellen Werten) angesprochen werden.

Beispiel :

```
class  Person
{  ...
  char name[20];                     //  Datenelement name
   ...

  Person(char* name)                 //  lokale Variable überdeckt Datenelement
  { strcpy(this->name,name);         //  mit this->name kann dann der Zugriff
  }                                  //  auf das Datenelement name erfolgen

  Person kopie()                     //  eigenes Objekt mit aktuellen Daten
  { return *this;                    //  als Rückgabewert ausgeben
  }

};
```

Destruktor :

Ein Destruktor ist für das Freigeben eines Objektes zuständig.
Im allgemeinen besteht keine Notwendigkeit einen expliziten Destruktor zu definieren, die korrekte Freigabe wird automatisch durch den Standarddestruktor ausgeführt.
Ein expliziter Destruktor ist dann notwendig, wenn dynamisch erzeugte Datenelemente einer Klasse wieder korrekt freigegeben werden müssen.
Ein expliziter Destruktor wird wie ein Konstruktor, jedoch mit einleitender Tilde (~) definiert.

Explizite Destruktordefinition :

```
<KlassenName>::~<KlassenName>()
{
}
```

Beispiel : Destruktor zur Klasse Person (nur zur Demonstration, ohne Notwendigkeit)

```
class  Person
{
   ...

    ~Person()                          // Destruktor
    { cout << " Destruktor von " << name ;
    }
   ...
};
```

abstrakte Klassen :

Eine abstrakte Klasse enthält zumindest eine **rein-virtuelle Methode** (pure-virtual).
Von einer abstrakten Klasse kann keine Instanz direkt erzeugt werden, sondern sie dient nur als Basisklasse und Vorlage für Methoden. In den abgeleiteten Klassen können die rein-virtuellen Methoden dann durch konkrete Methoden überschrieben werden.
Eine rein-virtuelle Methode hat keinen Definitionsteil und endet mit “=0“.

Definition einer rein-virtuellen Methode :

```
virtual <Typ><MethodenName>(Parameterliste) = 0 ;
```

Überladene Operatoren :

Die vordefinierten Operatorsymbole von C++ (arithmetische, logische, Vergleich, ...) können für Klassen neu definiert werden und damit neue Operationen ausführen.

Operatordefinition :

```
Returntyp operator Symbol( Parameter )
{  Anweisungen ;
}
```

Beispiel : komplexe Rechnung mit überladenen Operatoren

```
//-------------------------------------------------------------------------
//                       Klassenvereinbarung
class komplex
{ public:
      float re,im;                              // Datenelemente
  public:
      komplex(float re, float im );             // Konstruktoren
      komplex();

      float real();                             // Realteil ausgeben
      float imag();                             // Imaginärteil
      float betrag();                           // Betrag
      float phase();                            // Phase

      komplex operator+=(komplex &);            // überladene Operatoren
      komplex operator-=(komplex &);

////////////////////////////////////////////////////////////////////////
// überladene Operatoren mit zwei Parametern können nur über
// friend Funktionen ausgeführt werden !

      friend komplex operator+(komplex &, komplex &);
      friend komplex operator-(komplex &, komplex &);

      friend komplex operator*(komplex &, komplex &);
      friend komplex operator/(komplex &, komplex &);
       ...
};

//-------------------------------------------------------------------------
//                       externe Methoden Definitionen

komplex komplex::operator+=(komplex &z)
{  re += z.re; im += z.im;
   return *this;
}

komplex operator+(komplex &z1,komplex &z2)      // friend Funktion
{  return komplex(z1.re + z2.re, z1.im + z2.im);
}

//-------------------------------------------------------------------------
//                       Hauptprogramm
int main()
{  komplex z,z1(1.0,1.0),z2(-1.0,1.0),z3;       // Konstruktor Aufrufe
   z1+ = z2;                                    // komplexe Addition mit +=
   z3  = z1 + z2;                               // komplexe Addition mit +
    ...

}
```

Objekte mit dynamischen Datenelementen :

Bei Klassen mit dynamisch erzeugten Elementen sind folgende Regeln zu beachten :

- über den expliziten Konstruktor wird dynamischer Speicherplatz reserviert
- ein expliziter Kopierkonstruktor ist für eine korrekte tiefe Kopie notwendig
- der Zuweisungsoperator = ist als überladener Operator für eine tiefe Kopie zu definieren
- über einen expliziten Destruktor muß die korrekte Freigabe des dynamischen Speicherplatzes ausgeführt werden.

Beispiel : String Klasse

```
//////////////////////////////////////////////////////////////////
//    program name  :  String.cpp                               //
//       mit dynamischer Speicherverwaltung                     //
//       und überladenen Operatoren = + ==                      //
//////////////////////////////////////////////////////////////////
 ...
class String
{  private:
       char *ps;                                  // Zeiger auf dyn. String
       int l;                                     // Stringlänge

   public:
       String( char* );                           // Konstruktoren
       String( String &s );
       virtual ~String();                         // Destruktor

       int length(){return(l);}                   // Stringlänge zurückgeben
       void out(){puts(ps);}                      // String ausgeben
       String& operator = (String&);              // String Zuweisung mit =
       friend String& operator + (String&,String&);  // String Verkettung mit +
       friend int operator == (String&,String&);     // String Vergleich mit ==
};

inline String::String(char *s )                   // Konstruktor für C-String
{  l  = strlen(s);                                // Länge zuweisen
   ps = new char[l+1];                            // dyn. Speicherplatz
   strcpy(ps,s);                                  // s auf ps kopieren
}

inline String::String(String& s )                 // Copy-Konstruktor
{  l  = s.l;                                      // Länge zuweisen
   ps = new char[l+1];                            // dyn. Speicherplatz
   strcpy(ps,s.ps);                               // String s kopieren
}

inline String::~String()                          // Destruktor
{  delete[] ps;                                   // zur korrekten Freigabe
}

String& String::operator = (String& s)           // = Operator für tiefe Kopie
{  l  = s.l;                                      // Länge von s zuweisen
   delete[] ps;                                   // Speicherplatz freigeben
   ps = new char[l+1];                            // neuer Speicherplatz
   strcpy(ps,s.ps);                               // String s kopieren
   return (*this);                                // Objekt zurückgeben
}

 ...

int main()
{
   String s1("String1"),s2=s1;                    // Instanzieren
   s3=s1;                                         // String zuweisen
   ...
}
```

Templates :

Templates sind Schablonen für Funktionen oder Klassen. Sie erlauben Definitionen ohne der Angabe eines konkreten Datentyps. Schablonen enthalten alle Programmvorschriften mit einem Typplatzhalter und können dann für beliebige Datentypen verwendet werden. Schablonen sind damit auch eine Alternative zu überladenen Funktionen.

Funktions-Templates : **`template <class`** *`Typplatzhalter`*`>`
Funktionsdefinition mit Typplatzhalter

Beispiel : Funktion tauschen als Template

```
template <class Typ>                        // template Definition
                                            // mit Typ als Typplatzhalter
void tauschen (Typ &x, Typ &y)              // Funktion mit Typplatzhalter
{ Typ z=x;
  x=y;
  y=z;
}

 ...                                        // Aufruf der Funktion tauschen
                                            // mit beliebigen Datentypen
float x1,x2;
int   i1,i2;
tauschen(x1,x2);                            // Aufruf mit float - Argumenten
tauschen(i1,i2);                            // Aufruf mit int - Argumenten
```

Klassen -Templates : **`template <class`** *`Typplatzhalter`*`>`
Klassendefinition mit Typplatzhalter

Beispiel : Klasse vector(= eindim.Array) als Template

```
template <class Vect>
class vector
{ private:
      int dim;                              // Anzahl der Elemente
      Vect *start;                          // Zeiger auf Array-Anfang
  public:
      vector(int n);                        // Konstruktor
      ~vector() { delete [] start; }        // Destruktor
      void init(const Vect& v);             // Methode init
      int length() { return dim; }          // Methode length
      void redim(int n);                    // Methode redim
      Vect& operator[] (int index);         // Index Operator
      vector<Vect>& operator=(vector<Vect> &); // Zuweisungsoperator
};

template <class Vect>                       // externe Konstruktor Definition
inline vector<Vect>::vector(int n)
{  dim=x; start=new Vect[n];
}

template <class Vect>                       // externe Methoden Definition
void vector<Vect>::init(const Vect& v)
{  for (int i=0;i<dim;i++)
      start[i]=v;
}
```

```
template <class Vect>                        // redim Definition
void vector<Vect>::redim(int x)
{  Vect *p;
   int n;
   p = new Vect[x];                          // neuen Vektor erzeugen
   if ( x>dim ) n=x;                         // wer ist kleiner ?
   else n=dim;
   for (int i=0;i<n;i++)                     // Elemente kopieren
      p[i]=start[i];
   delete [] start;                          // alten Vektor freigeben
   start = p;                                // neue Adresse und
   dim = x;                                  // neue Dimension übernehmen
}

template <class Vect>                        // Indexoperator Definition
Vect& vector<Vect>::operator[] (int index)
{  if ((index>=0)&&(index<dim))
      return start[index];
}

template <class Vect>                        // Zuweisungsoperator Definition
vector<Vect>& vector<Vect>::operator=(vector<Vect> &v)
{  delete start;                             // alten Vektor freigeben
   dim = v.dim;                              // neue Dimension übernehmen
   start = new Vect[dim];                    // neuen Vektor erzeugen
   for (int i=0;i<dim;i++)                   // Elemente kopieren
     start[i]=v.start[i];
   return *this;                             // Kopie zurückgeben
}


void main()
{  cout << "                          Templates " << endl ;
   cout << "            oder Schablonen erleichtern Einiges  " << endl ;

   vector<int> a(10),b(5);                   // int - Vektoren instanzieren
   a.init(1);                                // Werte initialisieren
   b.init(2);
   for (int i=0;i<a.length();i++)            // alle Elemente a[i] ausgeben
       cout << setw(2) << a[i];

   a=b;                                      // Vektor - Zuweisung

   vector<float> x(5),y(5);                  // float - Vektoren instanzieren
   x.init(1.2);
   for (i=0;i<x.length();i++)
   {  y[i]=x[i];                             // Zuweisung y[i]=x[i]
      cout << setw(5) << y[i];
   }

   x.redim(10);                              // Vektor dynamisch vergrößern

   ...
 }
```

Schablonen können auch **mit mehreren Typplatzhaltern** erstellt werden :

```
template <class Typ1, class Typ2 >
Funktions- oder Klassendefinitionen mit Typ1, Typ2, ...
```