

# 1 Einführung in die Objektorientierte Programmierung OOP

Die objektorientierte Programmieretechnik wurde in den 80-er Jahren entwickelt und war der Ausweg aus der Software Krise. Die immer umfangreicheren Programme für grafische Benutzeroberflächen waren kaum mehr mit den klassischen prozeduralen Methoden beherrschbar.

Die OOP mit Klassenbibliotheken gehört heute zu den Standard-Werkzeugen zur Erstellung von Windows Programmen. (Visual C++ ).

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

Die **Modularisierung** (Zerlegungstechnik) sollte folgende Prinzipien erfüllen:

- Syntaktische Einheiten (Module) 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

Die **Wiederverwendbarkeit von Modulen** erfordert:

- Gemeinsamkeiten nutzen → Vererbung
- Darstellungsunabhängigkeit → Polymorphie

Die OOP erfüllt diese gestellten Anforderungen.

Die objektorientierte Programmierung in C wurde mit C++ im Jahr 1984 von Bjarne Stroustrup eingeführt. C++ enthält C als Untermenge und wird C immer mehr ablösen.

C++ ist heute eine der am meisten verwendeten Sprachen und wird auf vielen Plattformen, wie UNIX, MS-Windows eingesetzt. Neue Sprachentwicklungen (Java) lehnen sich auch sehr stark an die Sprache C++.

**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).

**Polymorphismus** ( polymorphism ) = Polymorphismus ist der Begriff für Vielgestaltigkeit und bedeutet in der OOP die Fähigkeit aus gleichnamigen Funktionen erst zur Laufzeit die richtige auszuwählen.

## 2 Klassen und Objekte

Der Grundgedanke der strukturierten Programmierung liegt darin, jedes Programm aus den drei Grundbausteinen Anweisung, Verzweigung und Schleife zusammenzusetzen.

Die strukturierte Programmierung kümmert sich aber nur um die Programmsteuerung, nicht um die Daten. In der objektorientierten Programmierung werden Daten und Programme als zusammengehörige Komponenten und als gleichberechtigte Partner verstanden: ein Objekt erhält nicht nur die Daten, die zu behandeln sind, sondern auch die Operationen, die auf diese Daten anzuwenden sind.

Die Bezeichnung "**objektorientiert**" lässt schon vermuten, dass **Objekte** ("**Instanzen**") die zentralen Bausteine des Denkansatzes sind. Dabei handelt es sich um Dinge in unserer Welt, die sich eindeutig definieren lassen und eine klare Abgrenzung zu anderen Objekten haben (z.B. "die graue Maus", "die schwarze Katze" usw.).

Auch Menschen und Tiere werden als Objekte aufgefasst. Wir beurteilen die Objekte unserer Welt nach zwei Kriterien:

1. Nach **statischen Merkmalen** (Eigenschaften), wie "**Farbe**" oder "**Form**".
2. Nach bestimmten **Verhaltensweisen** (Funktionen) die ein Objekt aufweist.  
So könnte das Verhalten einer Katze beispielsweise durch Funktionen, wie "**gehen**", "**fressen**" und "**schlafen**" modelliert werden.

Objektorientiert betrachtet, spricht man bei den

- Eigenschaften von "Attributen" oder "Instanzvariablen" und bei den
- Funktionen von "Methoden".

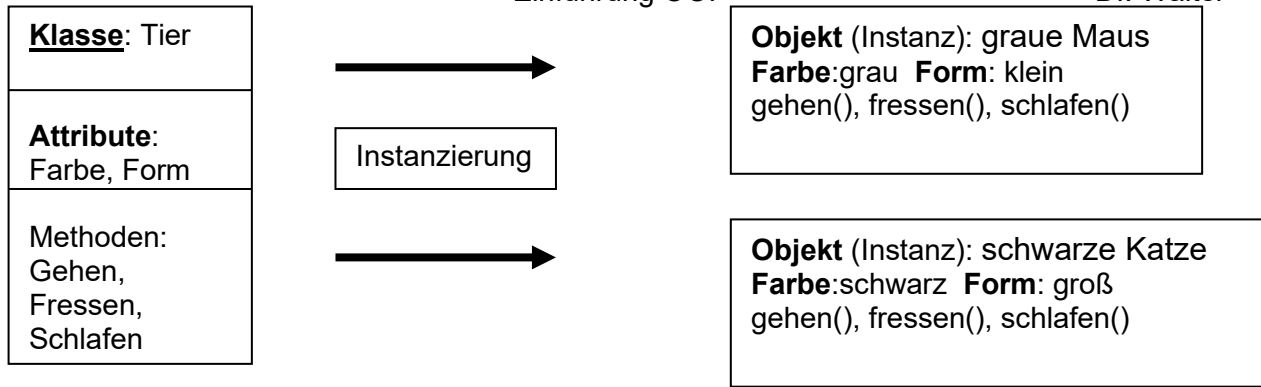
Wir haben die Fähigkeit, alle Tiere zu klassifizieren, sie also einer bestimmten "**Klasse**" (z.B. "**Tier**") zuzuordnen, und trotzdem jedes einzelne Tier als Individuum von anderen zu unterscheiden (z.B. "**Maus**", "**Katze**"). Wir wissen, dass alle Tiere gehen können, erkennen aber, dass eine Katze andere Eigenschaften (z.B. "**Farbe**", "**Form**") als eine Maus besitzt.

Diese Vorstellungswelt entspringt das Konzept der Klassen und ihrer Objekte in der objektorientierten Systementwicklung.

Es gibt Klassen (z.B. "**Tier**"), die man sich als eine Art Vorlage zur Erstellung konkreter Objekte (z.B. "**graue Maus**", "**schwarze Katze**") vorstellen kann. Durch sogenannte "**Instanziierung**" werden "**Objekte**" ("**Instanzen**") erstellt, deren Attribute und Methoden der Klasse entnommen sind. Während der Objekterstellung werden die jeweiligen Attribute wie Daten "gefüllt", z.B. bekommt die Eigenschaft "**Farbe**" der Katze den Wert "**schwarz**".

Die in der Klasse definierten Attribute existieren pro Instanz einmal und werden häufig auch als "**Instanzvariablen**" bezeichnet, d. h. Maus und Katze besitzen jeweils das Attribut "**Farbe**" wobei die Farbe der Maus völlig unabhängig von der Farbe der Katze ist.

Auch Methoden wie "**gehen**", "**fressen**" und "**schlafen**" kommen pro Instanz einmal vor. Eine Maus besitzt also eine Methode "**gehen**", die mit der "**gehen**" Methode einer Katze nichts zu tun hat.



Definitionen:

Die Gesamtheit von Objekten einer Art bezeichnet man als **Klasse**.

Ein **Klasse** fasst **Objekte** der gleichen Art zusammen!

Ein **Objekt** einer bestimmten Klasse ist eine Instanz dieser Klasse.

**Objekte** gleicher Art sind **Instanzen** der gleichen Klasse!

## Praktische Umsetzung in C++:

In C++ wird eine Klasse zur Definition von Objekten verwendet. Bei der Definition einer Klasse geht es darum, möglichst viele Informationen über ein Objekt direkt innerhalb einer Klasse einzubringen. Eine für ein bestimmtes Programm erstellte Klasse kann daher auch in anderen Programmen verwendet werden.

Eine C++ Klasse bewirkt, dass in einem Programm Daten und Funktionen, die mit diesen Daten arbeiten, zusammengefasst werden.

Ähnlich wie die Struktur muss auch eine Klasse einen eindeutigen Namen besitzen, gefolgt von einem oder mehreren Attributen, die in geschweiften Klammern stehen. Eine Klasse ist ein abstrakter Datentyp, der Daten und Methoden enthält.

```
class tier
{
    int gewicht;           // Daten Attribute (engl. Member)
    char name[20];

    void eingabe(int );    // Methode zur Eingabe
};
```

Nach der Definition einer Klasse können Variablen des Klassentyps (Objekte) deklariert werden. Jedes Objekt belegt dabei Speicherplatz !

```
tier hund, katze, maus;
```

In folgendem Beispiel wird die Klasse Mitarbeiter definiert:

```
class Mitarbeiter
{
    public:
        char Name[64];           //Daten Attribute der Klasse
        long Mitarbeiter_id;
        float Gehalt;

        void show_Mitarbeiter(void);
};

void Mitarbeiter::show_Mitarbeiter (void)
{
    cout << "Name: " << Name << endl;           // Funktionsdefinition
    cout << "Id: " << Mitarbeiter_id << endl;
    cout << "Gehalt: " << Gehalt << endl;
}
```

Die Methoden sind üblicherweise nur als Prototypen in der Klasse angegeben (**Deklaration**) und die **Methodendefinitionen** werden dann außerhalb der Klasse ausgeführt.

Eine komplette Funktionsdefinition innerhalb der Klasse wird als **Inline** Funktion ausgeführt und hat die Wirkung, dass bei der *Kompilierung* der Aufruf durch den Funktionskörper ersetzt wird. (vorteilhaft bei kurzen, rasch auszuführenden Funktionen)

Die Angabe der Klasse mit dem **Bereichsoperator "::"** vor dem Funktionsnamen ermöglicht den Zugriff auf alle Elemente der Klasse in dieser Funktion.

```
int main (void)
{
    Mitarbeiter Arbeiter, Boss;           // Objekte, Instanzen

    strcpy (Arbeiter.Name, "Ruderer");
    Arbeiter.Mitarbeiter_id = 12345;      // externer Zugriff
    Arbeiter.Gehalt = 2500;

    strcpy (Boss.Name, "Steuermann");
    Boss.Mitarbeiter_id = 101;
    Boss.Gehalt = 10000;

    Arbeiter.show_Mitarbeiter();

    Boss.show_Mitarbeiter();
}
```

Dieser externe Zugriff auf die Datenelemente und die Methoden ist nur für public-Elemente möglich!

### Datenkapselung (Zugriffskontrolle):

Elemente einer Klasse sind zu einer Einheit verbunden. Der Zugriff auf die Elemente einer Klasse kann mit folgenden Zugriffsspezifizierern gesteuert werden:

- **public:** Ein Programm kann von **jeder Funktion** aus direkt auf *public-Attribute* zugreifen.
- **private:**  
Auf *private-Attribute* hat das Programm nur mittels Klassenfunktionen (Methoden) zugreifen.  
Standardmäßig (wenn kein Spezifizierer angegeben wird) ist in C++ das Attribut *private* !
- **protected:**  
Diese Elemente sind geschützt jedoch auch in abgeleiteten Klassen zugreifbar.

## 3 Konstruktor:

Ein Konstruktor ist eine spezielle Methode der Klasse zur Initialisierung einer Instanz („**Instanzierung**“), d.h. ein Konstruktor wird benötigt, um Objekte einer Klasse zu initialisieren.

Der Konstruktor ist dadurch gekennzeichnet, dass er denselben Namen wie die Klasse hat.

```
class Person                                //   Klasse Person
{
    char name[20];                          //   Datenelemente (Member)
    .....                                  //   Standardkonstruktor
    Person( ) { }                           //   überladener Konstruktor
    Person(char *n)                         //
    {
        strcpy(name,n);
    }
    ....
};

int main( )
{
    Person Peter,Hans;    // Instanzierung über Standardkonstruktor
    Person Max("Maximilian"); // überladener Konstruktor
    ...
}
```

### Bemerkung:

- Wird kein Konstruktor definiert, so wird automatisch ein **Standardkonstruktor** ausgeführt. Dieser muss dann nicht explizit definiert werden.
- Wird mindestens ein überladener Konstruktor definiert, so muss auch der Standardkonstruktor explizit definiert werden, wenn dieser auch verwendet wird ( wie in diesem Beispiel)
- Ein Konstruktor hat **keinen** return Typ ! ( auch nicht void )
- Für eine Klasse können auch mehrere, überladene Konstruktoren erstellt werden, um Initialisierungen auf unterschiedliche Arten zu ermöglichen.

**Beispiel : Konstruktoren zur Klasse Person**

```

class Person                                // Klasse Person
{                                           // Datenelemente
    public:                                // öffentliche Datenelemente
        char name[20];
        int alter;
        char beruf[20];

    public:
        Person(){}                        // Standardkonstruktor
        Person(char *, int, char *);      // überladener Konstruktor
        Person(char *nam)                 // Konstruktor inline definiert
        { strcpy(name,nam);
        }

        void eingabe();                    // Methoden zur Daten-Eingabe
    ...
};

// externe Konstruktor Definition

Person::Person(char *nam,int alt,char *ber)
{
    strcpy(name,nam);
    alter=alt;
    strcpy(beruf,ber);
}

int main()
{
    Person mann;                           // Standardkonstruktor
    Person sohn("Peter",18,"Schueler");    // überladener Konstruktor
    Person frau("Fr.Huebsch");

    cout << sohn.alter;
    mann.eingabe();

    ...
}

```

## 4 Inline Elementfunktionen :

Inline Funktionen bzw. Methoden 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 (Inline-Definition)

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

- Deklaration in der Klasse und Definition außerhalb ( übersichtlicher)

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

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

Bemerkung: Inline-Definitionen müssen immer in der Datei erstellt werden, wo sie auch in der Klasse deklariert wurden (i.a. in der Header-Datei)

## 5 Default-Werte bei Konstruktoren

Bei Verwendung z.B. des Standardkonstruktors ist es möglich, Attribute einer Klasse mit Default-Werten zu versorgen.

```
class X
{
    int a;
    float b;
    ...
    X():a(3),b(17.3){} // Standardkonstruktor
}
```

oder einfacher:

```
class X
{
    int a;
    float b;
    ...
    X(){a=3; b=17.3;} // Standardkonstruktor
}
```

## 6 Überladene Methoden/Funktionen:

Überladene Methoden/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;
}
```

## 7 Statische Klassenelemente (static):

Jede Instanz einer Klasse legt die Datenelemente in eigenen Speicherplätzen an.

Ein **statisches Datenelement** hingegen wird nur einmal für alle Instanzen einer Klasse erzeugt und damit wird von verschiedenen Instanzen auf das gleiche Datenelement zugegriffen. Dies könnte auch mit globalen Variablen erreicht werden, widerspricht jedoch den Grundsätzen der objektorientierten Programmierung.

**Statische Funktionen** sind jene Funktionen, die statische Datenelemente verarbeiten.

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

    Teilnehmer(){ anzahl++; }         // Standardkonstruktor

    static int f_anzahl( )
    {
        return anzahl;               // statische Funktion
    }
};

int Teilnehmer::anzahl=0;              // einmalige Initialisierung

int main()
{
    Teilnehmer t1,t2;
    cout << t1.f_anzahl();           // Ausgabe: 2

    Teilnehmer t3;
    cout << t1.f_anzahl();           // Ausgabe: 3
    ...
}
```



## 8 Vererbung :

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

### 8.1 Beispiel - die abgeleitete Klasse Kunde:

```
class Kunde : public Person    // abgeleitete Klasse von Kunde
                               // mit Zugriffspez. public
{
    public:
        int kundennummer;
    private :
        float kontostand;
    public:
        void konto_eingabe( );
        void konto_abfrage( );
        Kunde(char *,int,char *,int);           // Konstruktor
};
```

#### Konstruktor bei abgeleitenden Klassen :

Der Konstruktor einer abgeleiteten Klasse kann Parameter als Argumente an den Basisklassen-Konstruktor weitergeben und eigene Datenelemente initialisieren.

**Beispiel** : Konstruktor der abgeleiteten Klasse Kunde

```
Kunde::Kunde(char *n,int alt,char *ber,int knr): Person(n,alt,ber)
{
    kundennummer=knr;
}
```

Konstruktor initialisiert eigene Datenelemente und ruft den Basisklassen-Konstruktor auf.

**Beispiel**: Zugriff auf Eigenschaften der Basisklasse

```
void Kunde::konto_abfrage( )
{
    cout << "Nr: " << kundennummer;    // eigenes Element
    cout << "Name: " << name;          // Element der Basisklasse
    cout << "Alter: " << alter;        // public Element
    cout << "Konto: " << kontostand;   // eigenes private Element
}
// ----- Hauptprogramm -----

void main
{
    int i;
    Kunde Zauberer("Gandalf",100,"Zauberer",1234); // Konstruktor
    ....
    Zauberer.ausgabe( );                          // Methode der Basisklasse
    strcpy(Zauberer.name,"Saruman"); // public Element
    Zauberer.konto_abfrage( );                     // Methode der eigenen Klasse
    ....
}
```

## 8.2 Vererbung von Zugriffs-Rechten:

Das Ableiten eine Subklasse von einer Basisklasse erfolgt nach folgender Syntax:

```
class Unter: zugriff Basis
```

**zugriff** ist der Ableitungs-Spezifizierer und gibt an, mit welchem Zugriffsrecht aus der Sicht des Anwenders oder weiterer abgeleiteten Klassen die Elemente der Basisklasse in die abgeleitete Klasse übernommen werden.

Die abgeleitete Klasse selbst hat immer mittels ihrer Methoden Zugriff auf alle *public* und *protected* Elemente ihrer Basisklasse!

Der Ableitungs-Spezifizierer **zugriff** begrenzt die maximalen Zugriffsmöglichkeiten in den abgeleiteten Klassen.

Die Voreinstellung (default) des Ableitungs-Spezifizierer ist bei Klassen **private**, bei Strukturen **public**.

### Überblick des Zugriffsspezifizierers:

Ableitungs-Spezifizierer	Member Basisklasse	Member abgeleit. Klasse
private	private	kein Zugriff
	protected	private
	public	private
protected	private	kein Zugriff
	protected	protected
	public	protected
public	private	kein Zugriff
	protected	protected
	public	public

### Beispiel: Zugriffsspezifizierer ist protected

```
class Tier          // Definition der Basisklasse
{
    public:    int pub;
    protected: int prot;
    private:   int priv;
};

class Fische : protected Tier          // Von Basis abgeleitete Klasse
{
    ....
};

class Forellen: public Fische          // Klasse von Fische ableiten
{
    ....
    void forelle_fangen()              // Methode von Forelle
    {
        pub = ....;                   // Zugriff ist erlaubt
        prot = ....;                  // Zugriff ist erlaubt
        priv = ....;                  // Zugriff ist nicht erlaubt
    }
}
```

```

    }
}
...
int main(...)
{
    Fische    fisch1;    // Objekt der Klasse Fische
    Forellen fisch2;    // Objekt der Klasse Forellen

    fisch1.pub  = ...;    // nicht erlaubt
    fisch1.prot = ...;    // nicht erlaubt
    fisch2.pub  = ...;    // nicht erlaubt

}

```

Damit können zwar die Memberfunktionen der Klassen **Fische** und **Forellen** immer noch auf das Element **pub** zugreifen, aber der direkte Zugriff darauf aus der Applikation heraus über Objekte von Typ **Fische** und **Forellen** ist gesperrt.

Ist der Ableitungs-Spezifizierer **private**, so werden alle vererbten Elemente der Basisklasse **Tier** zur **private** Elementen der abgeleiteten Klasse **Fische**.

Damit besitzt nur noch die Klasse **Fische** Zugriff auf die **public** und **protected** Elemente der Basisklasse **Tier**.

Selbst die Methoden von **Forellen** haben jetzt keinen Zugriff mehr auf die Member der Klasse **Tier**!

### 8.3 Mehrfach-Vererbung :

Eine abgeleitete Klasse erbt von mehreren Basisklassen.

```

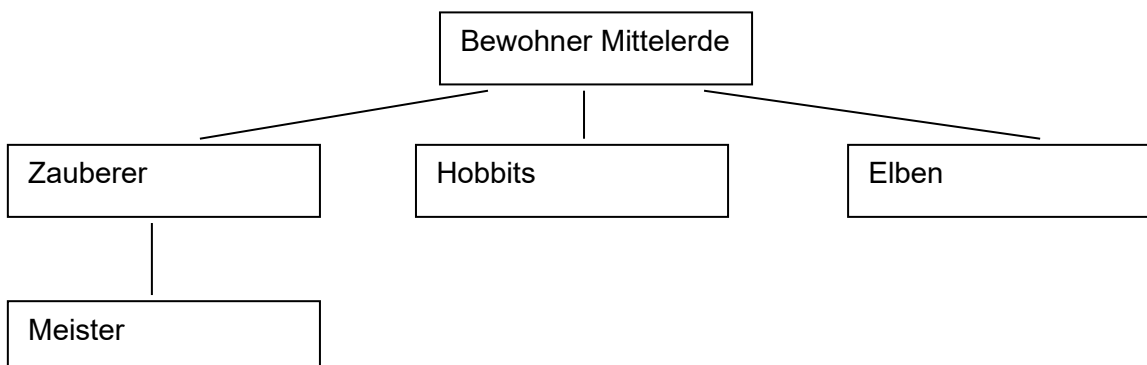
class <KlassenName> :<Spezifizierer> <Basisklasse1>,
                    <Spezifizierer> <Basisklasse2>,
                    <Spezifizierer> <Basisklasse3>
{
    < Datenelemente >
    < Methoden >
};

```

**Beispiel:** abgeleitete Klasse **Kind** erbt von Basisklassen **Mutter** und **Vater**

### 8.4 Klassen-Hierarchie :

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



## 9 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!

### Definition :

```
class <Klasse1>
{
    .....
    friend class <Klasse1_freund>;
    friend <Funktion_freund>;
    .....
};
```

Die Klasse <Klasse1\_freund> und die Funktion <Funktion\_freund> dürfen auf die geschützten Elemente der Klasse <Klassen1> zugreifen.

### Beispiel:

Zugriff einer fremden Funktion auf Elemente einer Klasse

```
class die_eine
{ private:
    int wert;
    friend void fremd (die_eine *,int);           // Zeigerparameter
    .....
};

void fremd (die_eine *z, int w)
{
    z->wert = w;
}

// Aufruf in main() :
....
die_eine beste;

fremd (&beste,100);
```

Der Wert 100 wird trotz private-Attribut mit der Funktion fremd verändert!

## 10 Polymorphie :

**Polymorphie** bedeutet in der OOP die Fähigkeit, aus mehreren Methoden mit gleicher Signatur aus einer Klassenhierarchie erst zur Laufzeit durch spätes Binden die richtige auszuwählen. Beim (normalen) frühen Binden wird bereits beim Übersetzungsvorgang die Adresse der Methode bei der Aufrufstelle festgelegt. Damit wird bei Objekten von abgeleiteten Klassen immer die Methode, die als erste vom Compiler gefunden wird, angesprochen. Bei der späten Bindung wird hingegen erst zur Laufzeit die Adresse aus einer Tabelle mit Referenzen (virtual method table) ausgewählt. Dynamisches oder spätes Binden wird über virtuelle Methoden ausgeführt.

**Virtuelle Methoden** werden mit dem Spezifizierer `virtual` deklariert und müssen in allen abgeleiteten Klassen die gleiche Signatur besitzen.

### Definition virtueller Methode :

`virtual <Typ><MethodenName> (Parameterliste)`

### Beispiel :

```
class Schueler : public Person           // abgeleitete Klasse Schueler
{ public:
    int katalognummer;

    virtual void nummer( );             // virtuelle Funktion nummer()
};

class Lehrer : public Person            // abgeleitete Klasse Lehrer
{ public:
    long personalnummer;

    virtual void nummer( );             // virtuelle Funktion nummer()
};

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
}
```

```
// Klasse Person mit virtuellen Methode nummer()

class Person
{ public:
    char name[20];
    .....

    void ausgabe( );          // Ausgabe der Daten

    virtual void nummer( ){} // virtuelle Funktion nummer()
};

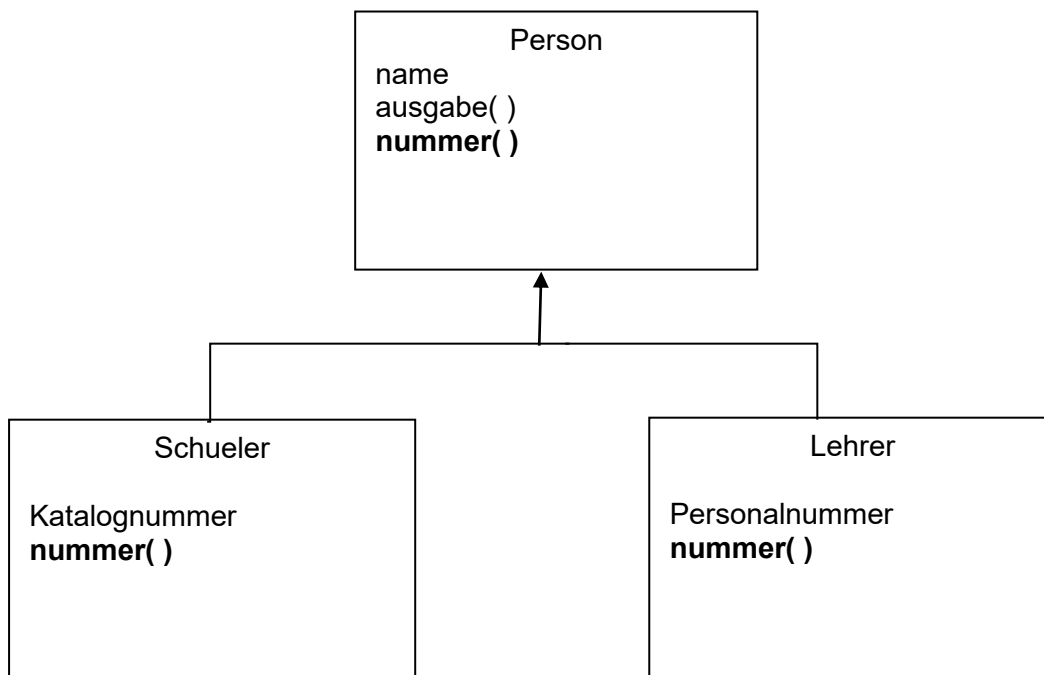
void Person::ausgabe( )
{
    printf("\n Name : %s",name);
    printf("\n Alter : %d",alter);
    printf("\n Beruf : %s",beruf);

    nummer( );          // Aufruf der virtuellen Funktion nummer( )
}
```

Die abgeleiteten Klassen **Schueler** und **Lehrer** haben die virtuelle Methode **nummer( )**. Die virtuelle Methode **Schueler::nummer( )** dient zur Ausgabe der Katalognummer, die virtuelle Methode **Lehrer::nummer( )** hingegen zur Ausgabe der Personalnummer. Die Methode **nummer( )** wird somit **vielgestaltig** (*polymorph*) eingesetzt.

Die Methode **nummer( )** wird dabei von der Methode **ausgabe()** der Basisklasse **Person** aufgerufen und ist daher in der Klassen- und Methodendefinition noch einzufügen.

#### Klassendiagramm:



## 11 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 Standarddestruitor ausgeführt.

Ein expliziter Destruktor ist immer dann notwendig, wenn dynamisch erzeugte Datenelemente einer Klasse wieder korrekt freigegeben werden müssen (-> sonst "hängende Referenz").

Ein expliziter Destruktor wird wie ein Konstruktor, jedoch mit einleitender Tilde (~) definiert.

### Beispiel:

```
class Person
{
    char *name;           // C-String als dynamisches Element
    ...
    // z.B. im Konstruktor
    ... name = new char[len];           // String soll Länge len haben

    ~Person( )             // Destruktor
    {
        delete [] name;
        cout << " Speicher wurde freigegeben! ";
    }
};
```

## 12 Dynamische Objekte:

Objekte können wie auch Variablen in C++ dynamisch mit **new** erzeugt werden.  
Dazu muß zuerst ein Zeiger auf die Klasse vereinbart werden. Der Zugriff auf das Objekt erfolgt dann über den Zeiger.

Die Freigabe des Speicherplatzes wird mit **delete** ausgeführt.

### Beispiel :

```
Person *p,*liste;           // Zeiger auf Klasse Person
p = new Person;             // dynamisches Objekt

p = new Person("Gandalf",22,"Zauberer"); // mit Konstruktor

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

p->eingabe( );              // Aufruf einer Methode

delete p;                   // 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
```

## 13 Zeiger this

Das Schlüsselwort „this“ stellt einen Zeiger auf das eigene Objekt einer Klasse dar.

Mit **this** kann daher innerhalb einer Methode auf die Instanz verwiesen werden, die an dieser Stelle noch nicht definiert ist. Mit **\*this** kann das ganze Objekt selbst angesprochen werden!

```
class person
{
    ...
    Person(char *name){strcpy(this->name, name);}

    // eigenes Objekt mit aktuellen Daten wird zurückgegeben
    Person copy(){ return *this; }

}
```

**this->name** ist gleichbedeutend mit dem direkten Zugriff auf das Attribut **name** der Klasse, wobei **name** hier auch als Übergabevariable verwendet wird! Der Zeiger **this** dient hier zur Unterscheidung von Klassen-Attribut und Übergabeparameter.

Verwendung: dynamische Listen, überladene Operatoren (siehe Kapitel 16)

## 14 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 Standard-Konstruktor
}

class Konstante3
{
    enum { K=20 };                   // int-Konstante über enum
}
```



## 15 Abstrakte Klassen:

Eine abstrakte Klasse enthält zumindest eine rein-virtuelle bzw. abstrakte Methode (**pure-virtual method**). Von einer abstrakten Klasse kann keine Instanz (Objekt) erzeugt werden, sondern sie dient nur als Basisklasse und Vorlage für Methoden.

Erst in den abgeleiteten Klassen müssen die rein-virtuellen Methoden dann durch konkrete Methoden „überschrieben“ (overwrite) werden.

Eine rein-virtuelle Methode hat keinen Definitionsteil und endet mit „=0“!

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

### Beispiel:

```
#include <string>      // string-Klasse

class vehicle          // abstrakte Klasse
{
    public:
        string text;

        // abstrakte Methode muss überschrieben werden
        virtual void sound() = 0;

        // virtuelle Methode kann überschrieben werden
        virtual void move()
        {
            cout << "Diese Methode kann ueberschrieben werden." << endl;
            cout << "Ich stehe still!" << endl;
        }

        // konkrete Methode in der abstrakten Klasse (hier Getter-Methode)
        string getText()
        {
            return this->text;
        }

        // Destruktor für jede abgeleitete Klasse erzwingen
        virtual ~vehicle() {};
};

class car : public vehicle
{
    public:
        void sound() // Implementierung der virtuellen Methode
        {
            this->text = "Brumm brumm brumm";
            cout << this->getText() << endl;
        }
        virtual void move() // Überschreiben möglich
        { cout << "Ich fahre gerade aus!" << endl;
        }

        ~car() // Implementierung des virtuellen Destruktor
        {
            cout << "Auto zerstoert!" << endl;
        }
};
```

```

class scooter : public vehicle
{
public:
    void sound() // Implementierung der virtuellen Methode
    {
        this->text = "Roll Roll Roll";
        cout << this->getText() << endl;
    }
    virtual void move()
    { cout << "Ich huepfe!" << endl;
    }

    ~scooter() // Implementierung des virtuellen Destruktor
    {
        cout << "Roller kaputt!" << endl;
    }
};

// Funktion mit abstrakter Klasse (nur als Referenz möglich!!!)
void start(vehicle &Vehicle)
{
    Vehicle.move();
}

int main(void)
{
    car ferrari;
    scooter blunt;

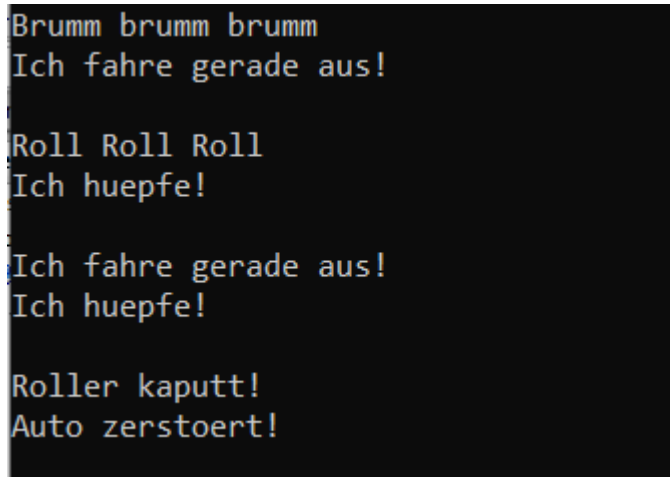
    ferrari.sound();
    ferrari.move();

    blunt.sound();
    blunt.move();

    start(ferrari);
    start(blunt);

    return 0;
}

```



```

Brumm brumm brumm
Ich fahre gerade aus!

Roll Roll Roll
Ich huepfe!

Ich fahre gerade aus!
Ich huepfe!

Roller kaputt!
Auto zerstoert!

```

**virtual void move()** in der abstrakten Klasse zeigt, wie Methoden deklariert und definiert werden können. Es steht dem Programmierer frei, ob er diese Methode später überschreibt oder nicht. ( Ausgabe, wenn nicht überschrieben: „Ich stehe still!“ )

Es können wie im Beispiel mit **getText()** auch Methoden vordefiniert und deklariert werden, die in der abgeleiteten Klasse nicht neu definiert werden müssen (Vererbung).

Eine weitere Eigenheit von C++ sind **Destruktoren**, die für abschließende Aufgaben wie Speicherfreigabe verwendet werden. Jede Klasse, deren Attribute nicht primitive Typen sind oder die andere Ressourcen verwendet (wie z. B. eine Datenbankverbindung), sollte diese unbedingt in ihrem Destruktor freigeben. Um immer auf den richtigen Destruktor zugreifen zu können, muss der Destruktor in der abstrakten Klasse als virtual deklariert sein.

## 16 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

```
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

// Operatoren mit einem Parameter

        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 &);
...
};

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

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

komplex operator*(komplex &z1, komplex &z2)
{ return komplex(z1.re*z2.re - z1.im*z2.im, z1.re*z2.im + z1.im*z2.re);
}

int main( )
{    komplex z1(1.2,3.6), z2(-5.5,7.2), z3;
    z2 += z1;
    z3 = z1+z2;
    ...
}
```

## 17 Objekte mit dynamischen Datenelementen:

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

- über den expliziten Konstruktor wird dynamisch Speicherplatz reserviert
- ein expliziter Kopierkonstruktor ist für eine korrekte tiefe Kopie bei der Instanzierung 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

```

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

public:
    String(){ps = NULL;}     // Standardkonstruktor
    String( const char *);   // überladene 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&); // Stringverkettung
    friend int operator == (String&,String&);  // Stringvergleich
};

inline String::String(const 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"),s3;           // Instanzieren
    String s2=s1;                      // Instanzieren und Initialisieren
    s3=s1;                             // String zuweisen
    ...
    return 0;
}
```

## 18 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
void tauschen (Typ &x, Typ &y)                       // mit Typ als Typplatzhalter
{ Typ z=x;                                           // Funktion mit Typplatzhalter
  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*, ...