

Überblick C++

<u>1. GRUNDLAGEN UND KONZEPTE DER OBJEKTORIENTIERTEN PROGRAMMIERUNG</u>	2
<u>2. KLASSEN UND OBJEKTE</u>	2
2.1 WAS IST EINE KLASSE?	2
2.2 WAS IST EIN OBJEKT?	2
2.3 KONSTRUKTOREN	3
COPY KONSTRUKTOR	3
2.4 DESTRUKTOR	4
2.5 MEMBERVARIABLEN UND –FUNKTIONEN	4
2.5.1 STATISCHE MEMBERVARIABLEN	4
2.5.2 STATISCHE MEMBERFUNKTIONEN	4
2.6 SPEICHERSCHUTZATTRIBUTE	5
2.7 FRIEND-FUNKTIONEN UND –KLASSEN	5
2.7.1 FRIEND-FUNKTIONEN	5
2.7.2 FRIEND-KLASSEN	6
2.8 INLINE-FUNKTIONEN	6
2.9 DER SCOPE RESOLUTION OPERATOR	7
<u>3. FUNCTION- UND OPERATOR-OVERLOADING</u>	7
3.1 FUNCTION-OVERLOADING	7
Ermitteln der Adresse einer überladenen Funktion	9
3.2 OPERATOR-OVERLOADING	10
<u>4. VERERBUNG</u>	11
Mehrfache Vererbung	12
Ausführung der Konstruktoren / Destruktoren	13
4.1 PROTECTED MEMBERS	13
protected-Vererbung	13
4.2 VIRTUELLE BASISKLASSEN	13
4.3 POLYMORPHIE	13
<u>5. VIRTUELLE FUNKTIONEN</u>	14
5.1 ALLGEMEINES	14
5.2 PURE VIRTUAL FUNCTIONS	15
5.3 v-TABLE	15
5.4 VIRTUELLE FUNKTIONEN VS. FUNCTION-OVERLOADING	15

1. Grundlagen und Konzepte der Objektorientierten Programmierung

Das Konzept der objektorientierten Programmierung (OOP) wurde entwickelt, um die immer komplexer werdenden Anwendungen überschaubarer zu machen. OOP vereint die Vorteile der strukturierten Programmierung (wie in C) mit neuen Möglichkeiten, die völlig andere Wege der Programmierung ermöglichen.

Grundsätzlich teilt die OOP das Problem in mehrere Teile auf, welche jeweils Code und Daten enthalten. Weiters werden diese Teile in eine hierarchische Struktur gebracht.

Die folgenden drei Eigenschaften weisen alle OO-Programmiersprachen auf:

- ✓ Kapselung
- ✓ Polymorphie
- ✓ Vererbung

Eine nähere Beschreibung zu diesen Themen folgt später am Beispiel C++.

2. Klassen und Objekte

2.1 Was ist eine Klasse?

Eine Klasse ist ein abstrakter Datentyp wie zum Beispiel Geschäftspartner, Material oder Transaktion. Sie enthält Attribute und Methoden.

Die Deklaration einer Klasse in C++ ähnelt der einer Struktur und sieht wie folgt aus:

```
class Person
{
private:
    int age;

public:
    void setAge(int a);
};
```

Die Schlüsselwörter "private" und "public" werden später noch erklärt.

Attribute beschreiben die Struktur einer Klasse oder vereinfacht ausgedrückt, die Daten. Methoden beschreiben das Verhalten einer Klasse. Mit Hilfe dieser werden Operationen, Dienste und Funktionen angeboten. **Eine Methode wird in C++ nicht wie in C durch ihren Namen eindeutig identifiziert, sondern durch ihren Prototyp.**

2.2 Was ist ein Objekt?

Objekte sind Instanzen (oder Ausprägungen) einer Klasse. Vereinfacht ausgedrückt: Man kann eine Klasse in etwa als Datentyp ansehen und ein Objekt als zugehörige Variable dieses Datentyps. Der Unterschied ist der, daß ein Objekt auch die zugehörigen Methoden umfaßt.

Wenn es nun zwei Objekte namens "Müller" und "Huber" der Klasse "Geschäftspartner" gibt, sind die Daten in diesen Objekten zwar unterschiedlich, aber die Methoden sind exakt die gleichen.

2.3 Konstruktoren

Konstruktoren sind Methoden, die automatisch bei der Erzeugung eines Objekts ablaufen und haben immer den gleichen Namen wie die Klasse. Eine Klasse hat immer einen Konstruktor (**Default Konstruktor**), man kann aber mehrere Konstruktoren schreiben, die verschiedene Aufgaben erfüllen. Einsatzgebiete für Konstruktoren sind:

- ✓ Initialisierung einer Klasse durch parametrisierte Konstruktoren
- ✓ Bereitstellen von dynamischem Speicherplatz
- ✓ Copy Konstruktor

Copy Konstruktor

Wenn ein Objekt dynamisch Speicher anlegt und man eine Initialisierung dieses Objekts anhand eines typgleichen Objektes durchführen will, folgt daraus das Problem, daß beide Objekte auf denselben Speicherbereich zugreifen, wobei aber die beiden Objekte lediglich die gleichen Inhalte haben sollten. Das gleiche Problem tritt bei zwei weiteren Fällen auf. Zum einen, wenn ein Objekt als Argument an eine Funktion übergeben wird um zum anderen, wenn ein Objekt ein Rückgabewert einer Funktion ist. Um dieses Problem zu lösen, bietet C++ den Copy Konstruktor. Ein Copy Konstruktor sieht folgendermaßen aus:

```
<Klassenname> (const <Klassenname> &o)
{
    // Eigenlicher Code des Konstruktors
}
```

Das folgende Beispiel beinhaltet alle 3 möglichen Varianten eines Konstruktors:

```
class array
{
    int *p;
    int size;
public:

    // normaler constructor
    array (int sz)
    {
        p = new int [sz];
        if (!p) exit (1);
        size = sz;
    }

    // copy constructor
    array (array &a)
    {
        int i;

        p = new int[a.size];
        if (!p) exit (1);
        for (i = 0; i < a.size; i++) p[i] = a.p[i];
        size = a.size;
    }
}
```

```
~array() { delete [] p; }; //Destructor  
};
```

2.4 Destruktor

Destrukturen sind Methoden, die bei der Zerstörung eines Objekts automatisch aufgerufen werden und werden z.B. zum Freigeben von dynamisch allokierten Speicher oder zum Schließen einer Datei verwendet. Ein Destruktor sieht folgendermaßen aus:

```
~<Klassenname>()  
{  
    //eigentlicher Code  
}
```

2.5 Membervariablen und –funktionen

2.5.1 Statische Membervariablen

Wenn eine Membervariable einer Klasse als **static** deklariert wird, haben alle Objekte dieser Klasse Zugriff auf die gleiche Variable, da sie nur einmal im Speicher existiert. Weiters muß noch eine globale Definition für die Variable gemacht werden, da die Deklaration in der Klasse noch keinen Speicherplatz allokiert. Das folgende Beispiel veranschaulicht den Vorgang:

```
class shared  
{  
public:  
    static int a;                // Deklaration von a  
};  
  
int shared::a;                  // globale Definition von a  
  
main()  
{  
    // Initialisierung, bevor ein Objekt existiert  
    shared::a = 99;  
  
    cout << "Startwert von a: " << shared::a;  
    shared x;  
    cout << "Das ist x.a: " << x.a;  
}
```

Das Problem von statischen Membervariablen in Bezug auf OOP ist, daß sie immer das Prinzip der Abkapselung verletzen.

2.5.2 Statische Memberfunktionen

Bei statischen Memberfunktionen gibt es folgende Einschränkungen:

1. Sie haben nur Zugriff auf statische Members der Klasse (und natürlich auf globale Funktionen und Daten)
2. Sie haben keinen **this-Zeiger**
3. Es darf keine statische Version einer Funktion neben eine nicht-statischen existieren

Ein wichtiges Anwendungsgebiet für Statische Memberfunktionen ist die Initialisierung von statischen Daten einer Klasse bevor ein Objekt erzeugt wurde.

Beispiel f. die Deklaration einer statischen Memberfunktion:

```
class myclass
{
    static int resource;
public:
    static int get_resource();
};
```

2.6 Speicherschutzattribute

Da die OOP auf dem Prinzip der Kapselung beruht, sollen nicht alle Funktionen bzw. Daten nach außen hin sichtbar sein. Deshalb gibt es in C++ 3 Speicherschutzattribute:

public: Die entsprechende Komponente (Daten, Funktionen, Ereignisse) ist für alle Klassen sichtbar

protected: Die entsprechende Komponente ist für die Klasse selbst und für Erben sichtbar (s. Kapitel 4 – Vererbung)

private: Die entsprechende Komponente ist nur für die Klasse selbst sichtbar

Die Verwendung dieser Speicherschutzattribute wird oft auch als "**Information Hiding**" bezeichnet.

2.7 Friend-Funktionen und -Klassen

2.7.1 Friend-Funktionen

Friend-Funktionen werden verwendet, um Funktionen, die nicht Mitglied einer Klasse sind, Zugriff auf **private oder protected** Members dieser Klasse zu verschaffen. Die Deklaration erfolgt dadurch, daß der Prototyp der "fremden" Funktion in der Klasse deklariert wird. Dies sieht z.B. so aus:

```
class myclass
{
    int a, b;
public:
    friend int sum(myclass x);
    void set_ab(int i, int j);
};
```

2.7.2 Friend-Klassen

Wenn eine Klasse K1 Friend einer anderen Klasse K2 ist, so hat K1 Zugriff auf die als **private** deklarierten Bezeichner (z.B.: enum's, int, float, ...).

Eine Friend-Klasse hat **nur Zugriff** auf die Bezeichner der anderen Klasse, sie ist **nicht von der ursprünglichen Klasse abgeleitet**.

Beispiel:

```
class coins
{
    enum units {penny, nickel, dime, quarter, half_dollar};
    friend class amount;
};

class amount
{
    coins::units money;
public:
    int getm();
};
```

In diesem Beispiel hat die Klasse "amount" Zugriff auf "units"-enumeration der Klasse "coins", weil "amount" eine Friend-Klasse von "coins" ist.

2.8 Inline-Funktionen

Inline-Funktionen sind Makros sehr ähnlich, da der Code einer Inline-Funktion einfach an der Stelle, wo sie aufgerufen wird, vom Compiler eingefügt wird, d.h. es wird die zeitraubende Prozedur der Parameterübergabe über den Stack, usw. gespart. Daher ist eine Inline-Definition einer Funktion nur bei sehr einfachen und kurzen Aufgaben sinnvoll. Die Definition erfolgt mit Hilfe des Schlüsselwortes "**inline**".

Beispiel:

```
inline int max(int a, int b)
{
    return (a > b) ? a : b;
}

main()
{
    cout << max(10, 20);
    return 0;
}
```

Der Compiler wandelt dieses Programm dann folgendermaßen um:

```
main()
```

```
{
    cout << ((a > b) ? a : b);
    return 0;
}
```

In C++ werden Inline-Funktionen meistens für kurze Funktionsdefinitionen – und zwar gleich in der Klassendeklaration – verwendet. Der Compiler wandelt diese Funktionsdefinition dann automatisch in eine Inline-Funktion um. Hierbei ist die Angabe des Schlüsselwortes **“inline”** nicht zwingend.

Beispiel:

```
class myclass
{
    int a, b;
public:
    void init(int i, int j) { a = i; b = j; }
    void show() { cout << a << " " << b << "\n"; }
};
```

2.9 Der Scope Resolution Operator

Der Scope Resolution Operator (::) wird verwendet, um dem Compiler mitzuteilen, welcher Klasse das Member zugeordnet ist. Weiters wird der Scope Resolution Operator auch verwendet, um Zugriff auf “übergeordnete” Members zu haben, wenn ein lokaler Bezeichner mit dem gleichen Namen existiert.

Beispiel:

```
.
.
int i;          // globale Deklaration

void f()
{
    int i;     // lokale Deklaration

    i = 5;     // lokales "i" hat jetzt den Wert 5
    ::i = 10; // globales "i" hat jetzt den Wert 10
}
```

3. Function- und Operator-Overloading

3.1 Function-Overloading

Function Overloading ist grundsätzlich das Verwenden von gleichen Funktionen mit verschiedenen Parametern bzw. mit einer unterschiedlichen Anzahl von Parametern.

Weiters ist zu beachten, daß **zwei Funktionen mit gleichem Namen und gleicher Parameterliste aber unterschiedlichen Rückgabewerten nicht überladen** werden können.

Außerdem gibt es Situationen, in denen der Compiler nicht weiß, welche von zwei bestehenden Funktionen verwendet werden soll. Dies kann folgende Gründe haben:

1. Der Compiler führt eine automatische Typkonvertierung durch (z.B. von char auf double)
2. Bei Verwendung von Default-Argumenten
3. Zwei Funktionen sind gleich bis auf die Art der Parameterübergabe (Call by Reference bzw. Call by Value)

Beispiel:

ad 1)

```
float myfunc(float i);
double myfunc(double i);

main()
{
    cout << myfunc(10);    // Compiler weiß nicht, welche Fkt.
    return 0;              // er verwenden soll, da er 10 auf
}                          // float und auf double konvertieren
                          // kann

float myfunc(float i) { return i; }
double myfunc(double i) { return i; }
```

ad 2)

```
int myfunc(int i);
int myfunc(int i, int j=1);

main()
{
    cout << myfunc(10);    // Compiler weiß nicht, welche Fkt.
    return 0;              // er verwenden soll, da die zweite
}                          // zweite Möglichkeit durch den
                          // Default-Parameter auch mögl. wäre

int myfunc(int i) { return i; }
int myfunc(int i, int j) { return i*j; }
```

ad 3)

```
int f(int x);
int f(int &x);

main()
{
    int a = 10;
```

```

    f(a);           // Compiler weiß nicht, ob er "a" als
    return 0;      // Referenz oder als Wert interpretie-
}                // ren soll

int f(int x) { cout << "Function int f(int x)\n"; }
int f(int &x) { cout << "Function int f(int &x)\n"; }

```

Function-Overloading kommt auch oft bei Konstruktoren vor, wenn Objekte auf verschiedene Arten initialisierbar sein sollen.

Beispiel:

```

class date
{
    int day, month, year;
public:
    date(char *d) { sscanf(d, "%d%*c%d%*c%d", &month, &day, &year); }
    date(int m, int d, int y) { day=d; month=m; year=y; }
    void show_date();
};

```

Ermitteln der Adresse einer überladenen Funktion

Es kommt ab und zu vor, daß ein Programm über die Adresse einer Funktion auf diese zugreift. Dies sieht in C unter Betracht einer Funktion `myfunc()` und einem Zeiger `p` folgendermaßen aus:

```
p = myfunc;
```

Bei überladenen Funktionen wird dies komplexer, da der Compiler eine Funktion nicht anhand ihres Namens eindeutig identifizieren kann. Um die Möglichkeit dennoch zu nutzen muß man zuerst eine Zeigervariable in folgender Form deklarieren:

```
int (*<name_des_zeigers>)(parameterliste)
```

Danach kann die Funktionsadresse wie bisher der Zeigervariable zugewiesen werden.

Beispiel:

```

int myfunc(int a);
int myfunc(int a, int b);

main()
{
    int (*fp)(int a);           // Zeiger auf int xxx(int a)
    fp = myfunc;               // zeigt auf int myfunc(int a)
    return 0;
}

```

Hätte man `*fp` folgendermaßen deklariert: `int (*fp)(int a, int b)`, dann würde `fp` auf die Funktion `int myfunc(int a, int b)` zeigen.

3.2 Operator-Overloading

In C++ gibt es die Möglichkeit, fast jeden Operator (Ausnahmen: `.`, `::`, `.*`, `?`) mit einer, speziell auf die eigene Klasse versehene, Funktion zu versehen. Dies geschieht durch das Überladen einer **operator** Funktion. Diese Funktionen können – müssen aber nicht – Member der Klasse sein. Funktionen, die nicht zur Klasse gehören sind meistens **Friend**-Funktionen.

Man unterscheidet auch zwischen binären und unären Operatoren, da unäre Operator-Funktionen keinen Parameter übernehmen, binäre Operatoren hingegen einen.

Beispiel für einen binären Operator:

```
loc loc::operator+(loc op2)
{
    loc temp;
    temp.longitude = op2.longitude + longitude;
    temp.latitude = op2.latitude + latitude;
    reurn temp;
}
```

Aufruf:
`ob1 = ob1 + ob2;`

Der zweite, für die Verarbeitung notwendige Parameter, wird als **this-Zeiger** automatisch mitübergeben. Der Parameter, welcher der Funktion übergeben wurde, ist derjenige, der rechts vom Operator steht. Das Objekt, das links vom Operator steht, ist dasjenige, das die Operator-Funktion ausführt.

Um das Ergebnis der Operation – wie oben gezeigt – wieder einem Objekt der gleichen Klasse zuweisen zu können, muß der Rückgabewert auch stimmen.

Beispiel für einen unären Operator:

```
loc loc::operator++()
{
    longitude++;
    latitude++;
    return *this;
}
```

Wenn man den Operator `"="` überschreiben will, sollte man darauf achten, daß die Funktion `*this` zurückgibt. Dies erlaubt dann mehrfache Zuweisungen wie

```
ob1 = ob2 = ob3;
```

Wenn Friend-Funktionen zum Zweck des Operator-Overloading verwendet werden, müssen zwei Parameter übergeben werden, weil die Funktion nicht Member der Klasse ist und somit keinen **this-Zeiger** erhält.

Der **Vorteil von Friend-Funktionen zum Zweck des Operator-Overloading** ist es, daß die Reihenfolge der Operanden keine Rolle spielt. Ein Beispiel dafür:

Angenommen, "ob1" und "ob2" sind Objekte der Klasse "Bruch". Somit ist

ob2 = 100 + ob1 genauso möglich wie ob2 = ob1 + 100

Bei Verwendung von Member-Funktionen ist dies nicht möglich, da immer das links vom Operator stehende Objekt die überladene Funktion bereitstellen muß.

4. Vererbung

Die Vererbung ist eine der Eckpfeiler von OOP und wird von C++ in vollem Umfang unterstützt. Man kann Vererbung folgendermaßen definieren:

*"Die **Vererbung** definiert die Beziehung zwischen Klassen, in der eine Klasse (Subklasse oder Unterklasse) die Struktur und das Verhalten teilt, das in einer oder mehreren anderen Klassen (Superklassen oder Oberklassen) definiert wurde."*

Die Vererbung ermöglicht es, eine Basisklasse zu erstellen, die allgemeine Elemente enthält. Des Weiteren können mehrere abgeleitete Klassen erstellt werden, die alle Elemente der Basisklasse erben und ihre eigenen speziellen Merkmale hinzufügen.

Weiters ist es möglich eine abgeleitete Klasse als Basisklassen für andere abgeleitete Klassen zu verwenden. Daraus entsteht dann die Mehrfachvererbung.

Beispiel:

```
class base
{
    int i;
public:
    void set(int a) { i = a; }
    void show() { cout << i; }
};

class derived : public base
{
    int k;
public:
    derived(int x) { k = x; }
    void showk() { cout << k; }
};
```

Wenn eine Klasse als **public** vererbt wird, so werden alle public-Members und protected-Members der Basisklasse zu public-Members bzw. protected-Members der abgeleiteten Klasse. Die private-Members sind ausschließlich in der jeweiligen Klasse sichtbar.

Mehrfache Vererbung

In C++ gibt es außerdem die Möglichkeit, eine Klasse von mehreren Basisklassen abzuleiten.

Beispiel:

```
class base1
{
protected:
    int x;
public:
    void showx() { cout << x; }
};

class base2
{
protected:
    int y;
public:
    void showy() { cout << y; }
};

class derived : public base1, public base2
{
public:
    void set(int i, int j) { x = i; y = j; }
};

main()
{
    derived ob;
    ob.set(10, 20);    // von derived zur Verfügung gestellt
    ob.showx();       // von base1 zur Verfügung gestellt
    ob.showy();       // von base2 zur Verfügung gestellt
    return 0;
}
```

Diese Technik resultiert aber in folgendem Problem:

Klasse A hat Datenmember x

Klasse B und C sind von A abgeleitet und haben eigenen Datenmembers

Klasse D ist von B und von C abgeleitet

Im Programm wird nun ein Objekt des Typs D angelegt und auf die Variable x zugegriffen. Von welcher Klasse soll nun die Variable verwendet werden?

Es gibt 2 Möglichkeiten zur Lösung dieses Problems:

1. Scope Resolution Operator (siehe oben)
2. Virtuelle Basisklassen (siehe später)

Die Reihenfolge der Ausführung von Konstruktoren und Destruktoren im Falls der Vererbung sieht folgendermaßen aus.

Es wird der Konstruktor der Basisklasse aufgerufen, dann der der von der Basisklasse abgeleitete Konstruktor, usw. Bei den Destruktoren wird genau am anderen Ende der Hierachiestufen mit der Ausführung begonnen.

4.1 Protected Members

Diese haben grundsätzlich ähnlich Eigenschaften wie private-Members. Sie sind nur innerhalb einer Klasse sichtbar, außer diese Klasse wird vererbt. Wenn der Fall auftritt, daß eine Klasse public vererbt wird, so kann die abgeleitete Klasse auf die protected-Member der Basisklasse zugreifen.

protected-Vererbung

Wenn eine Klasse **protected** vererbt wird, so werden alle public- und protected-Members der Basisklasse zu protected-Members der abgeleiteten Klasse.

4.2 Virtuelle Basisklassen

Diese dienen zur Lösung des bei mehrfacher Vererbung entstandenen Problems (siehe oben). Dieser Lösungsansatz basiert darauf, daß die Basisklasse mit dem Schlüsselwort **virtual** versehen wird, wenn sie vererbt wird. Wenn nun eine mehrfache Vererbung wie im obigen Beispiel vorkommt, beinhaltet die Klasse D nur **eine** Kopie des Basisobjekts.

Beispiel für die Deklaration der Klasse B in Bezug auf das oben genannte Beispiel:

```
class B : virtual public A
{
    int k;
public:
    void showk() { cout << k; }
};
```

4.3 Polymorphie

C++ unterstützt Polymorphie zu Compilezeit und zu Laufzeit. Ersteres wird – wie schon besprochen – von Function- bzw. Operator-Overloading zur Verfügung gestellt, zweiteres wird von Vererbung und Virtuellen Funktionen (Function Overwriting) geboten, die im nachstehenden Kapitel noch beschrieben werden.

5. Virtuelle Funktionen

5.1 Allgemeines

Eine virtuelle Funktion wird in der Basisklasse mit dem Schlüsselwort **virtual** deklariert und in den abgeleiteten Klassen wieder definiert. Diese "Wiederdefinition" überschreibt die der Basisklasse. Generell kann man sagen, daß die Deklaration in der Basisklasse als "Platzhalter" für eine generelle Funktion fungiert, die in den abgeleiteten Klassen erst wirklich implementiert werden, weil diese unterschiedlich Aufgaben haben.

Sehr nützlich sind virtuelle Funktionen beim Erstellen von Containern, die unterschiedliche Objekte enthalten, aber alle einen gewissen Satz von Standardfunktionalität haben soll. Ein Beispiel wäre dafür die Basisklasse *Zeichenobjekt*, von dem die Klassen *Kreis* und *Rechteck* abgeleitet sind. Nun wird eine Liste von Zeichenobjekten erstellt, und ein Aufruf der Funktion *draw()* der Basisklasse soll dann je nachdem, ob das Objekt ein Kreis oder ein Rechteck ist, richtig gezeichnet werden.

Das wichtige bei virtuellen Funktionen ist ihr Verhalten, wenn sie via Zeiger aufgerufen werden. Wenn ein Basisklassenzeiger auf ein abgeleitetes Objekt zeigt, das eine virtuelle Funktion enthält, führt C++ die richtige Version dieser Funktion aus.

Virtuelle Funktionen können auch vererbt werden, d.h. sie können in mehreren Ebenen der Vererbungshierarchie immer wieder überschrieben werden.

Außerdem ist zu beachten, daß die virtuelle Funktion der Basisklasse ausgeführt wird, wenn diese in einer abgeleiteten Klasse nicht überschrieben wurde.

Beispiel:

```
class base
{
public:
    virtual void vfunc() { cout << "Base's vfunc()"; }
};

class derived1 : public base
{
public:
    void vfunc() { cout << "derived1's vfunc()"; }
}

class derived2 : public base
{
public:
    void vfunc() { cout << "derived2's vfunc()"; }
}

main()
{
    base *p, b;
    derived1 d1;
```

```
derived2 d2;

p = &b;
p->vfunc(); // vfunc() von base wird ausgeführt

p = &d2;
p->vfunc(); // vfunc() von derived2 wird ausgeführt

return 0;
}
```

5.2 Pure Virtual Functions

Eine Pure Virtual Function ist eine virtuelle Funktion, die in der Basisklasse nicht definiert wird. Die Deklaration sieht folgendermaßen aus:

```
virtual <type> <func_name>(<parameter_list>)=0;
```

Wie oben schon erwähnt wird die virtuelle Funktion der Basisklasse ausgeführt, falls in der abgeleiteten Klasse diese Funktion nicht überschrieben wird. Da dies in manchen Fällen keinen Sinn ergibt, kann man die Ausführung der virtuellen Funktion der abgeleiteten Klasse hiermit erzwingen. Falls diese Funktion nicht definiert ist, wird ein Compile-Fehler gemeldet.

5.3 v-Table

Bei Verwendung von virtuellen Funktionen wird erst zu Laufzeit entschieden, welche Definition der Funktion nun wirklich verwendet wird, da, wie schon oben erwähnt, aufgrund der Polymorphie mehrere Definitionen existieren können. Welche Funktion nun aufgerufen wird, hängt vom Typ des Objekts ab, welches gerade referenziert wird.

Dieser Mechanismus wird in C++ durch die v-Table realisiert. Wenn eine Klasse virtuelle Funktionen enthält, dann wird bei der Erzeugung eines Objekts dieser Klasse vom Konstruktor automatisch ein Zeiger auf diese – für uns nicht sichtbare - v-Table gespeichert. Diese v-Table enthält nun eine Liste von Zeigern auf alle virtuellen Funktionen dieser Klasse. Weiters ist zu erwähnen, daß sich alle Objekte einer Klasse dieselbe v-Table teilen und nicht-virtuelle Funktionen nicht in der v-Table aufscheinen.

5.4 Virtuelle Funktionen vs. Function-Overloading

Der Unterschied zwischen virtuellen Funktionen und Function-Overloading besteht darin, daß **virtuelle Funktionen exakt den gleichen Prototyp** haben müssen, was bei Function-Overloading nicht erlaubt ist.

Weiters ist zu beachten, daß virtuelle Funktionen in ihrer Klasse als **nicht-static** deklariert werden müssen.

Außerdem können sie **keine Friends** sein; Konstruktoren können nicht als virtual deklariert werden, Destruktoren hingegen schon.