# 3 Klassen

Im Folgenden werden wir das im vorigen Kapitel angeeignete theoretische Wissen in C++ umsetzen. Eine wesentliche Form der Abstraktion ist die Klasse, mit deren Besprechung wir nun beginnen werden.

## 3.1 Klassen und Attribute

Als erstes Beispiel wollen wir das in Kapitel 2 modellierte Schwein als Klasse formulieren. Klassen und Strukturen sind in C++ sehr ähnlich, weswegen wir uns zuerst einmal das Schwein als Struktur anschauen:

```
struct sSchwein
{
  int groesse;
  int gewicht;
  int saettigung;
};
```

Dies dürfte Ihnen vom Grundlagen-Kapitel bekannt sein. Wollten Sie nun eine Variable vom Typ *sSchwein* definieren, geschieht dies wie folgt:

```
sSchwein testschwein;
```

Auf einzelne Elemente der Variablen wird mit dem .-Operator zugegriffen:

```
testschwein.gewicht=40;
```

Eine Klasse sieht nun ganz genauso aus, nur dass anstelle des Schlüsselwortes *struct* das Schlüsselwort *class* steht. Hier einmal ein vollständiges Beispiel:

```
struct sSchwein {
  int groesse;
  int gewicht;
  int saettigung;
};

class kSchwein
{
  int groesse;
  int gewicht;
  int saettigung;
};

int main()
```

```
{

  sSchwein sschwein;
  kSchwein kschwein;

  sschwein.groesse=62;
  kschwein.groesse=75;
}
```

Den Quellcode dieses Beispiels finden Sie auf der CD unter /BUCH/KAP03/BSP001.CPP.

Es wird die vorher vorgestellte Struktur *sSchwein* deklariert sowie die Klasse *kSchwein*, deren Atttribute den Elementen von *sSchwein* entsprechen. Dann wird eine Variable *sschwein* vom Typ *sSchwein* und eine Variable *kschwein* vom Typ *kSchwein* definiert. Danach wird die Größe von *sschwein* auf 62 und die Größe von *kschwein* auf 75 gesetzt.

So weit, so gut. Bevor Sie weiterlesen, sollten Sie jedoch dieses Programm einmal kompilieren. Sie werden überrascht sein.

## 3.2 Öffentliche und private Attribute

Bei der Kompilation des obigen Beispiels werden Sie vermutlich eine Fehlermeldung erhalten, die einen Wortlaut wie »cannot access private member« oder ähnlich enthält.

Wenn Sie sich noch einmal den Abschnitt über Datenkapselung in Erinnerung rufen, wird Ihnen einiges klarer. Dort hieß es, dass das Optimum der Datenkapselung erreicht ist, wenn man von außen nicht mehr auf die Attribute einer Klasse zugreifen kann. Und genau dieses Optimum hat unser Beispiel erreicht. Wir können dem Attribut *groesse* von *kSchwein* keinen Wert zuweisen, weil wir dies von außen versuchen.

Wenn nicht anders definiert, sind Attribute einer Klasse privat.

**public**

Nun kann es aber in einigen Ausnahmen nützlich sein, wenn diese strikte Forderung der Datenkapselung gelockert wird. Daher gibt es in C++ die Möglichkeit, die Datenkapselung komplett aufzuheben. Dies erreicht man dadurch, dass man die Attribute als öffentlich deklariert. Dazu benutzen wir das Schlüsselwort **public**, welches auf Deutsch soviel wie »öffentlich« heißt. Hier nun die abgeänderte Klasse *kSchwein*:

```
class kSchwein
{
  public:
  int groesse;
```

```
  int gewicht;
  int saettigung;
};
```

Mit dieser Klassendefinition wird sich unser Beispiel von vorhin erfolgreich kompilieren lassen.

**private**

Die bloße Wahl zwischen entweder »Alle Attribute privat« oder »Alle Attribute öffentlich« wäre eine sehr starke Einschränkung, deswegen können die Zugriffsrechte auch gemischt werden. Um einen privaten Abschnitt – also einen gekapselten Abschnitt – einzuleiten, wird das Schlüsselwort **private** verwendet, was auf Deutsch soviel wie »privat« heißt:

```
class beispiel
{
  private:
  int a;
  float b;

  public:
  char c[8];
  int d;
  double e;

  private:
  unsigned int f;
  long g;

  public:
  short h;
};
```

**Zusammenfassen der Zugriffsrechte zu Gruppen**

Sie sehen, dass Sie die Schlüsselwörter *private* und *public* in beliebiger Reihenfolge und beliebig oft benutzen können. Sie sollten sich jedoch der Übersicht halber angewöhnen, immer Attribute mit gleichem Zugriffsrecht zu einer Gruppe zusammenzufassen und mit der Gruppe der privaten Attribute zu beginnen.

Fassen Sie immer Attribute mit gleichem Zugriffsrecht zusammen und beginnen Sie mit dem kleinsten Zugriffsrecht.

```
class beispiel
{
  private:
  int a;
  float b;
  unsigned int f;
  long g;
```

```
  public:
  char c[8];
  int d;
  double e;
  short h;
};
```

Doch warum sollte man überhaupt Attribute als privat deklarieren, wenn man sowieso nicht an sie herankommt? Ganz einfach: Wir benutzen Methoden.

## 3.3 Methoden

Wie wir vorher schon besprochen haben, sind Methoden die zu einer Klasse gehörigen Funktionen[1]. Und gerade weil die Methoden zu einer Klasse gehören, können sie auch auf die privaten Attribute der Klasse zugreifen.

**Methodendeklaration**

Hier ein Beispiel einer Klassendefinition:

```
class testklasse

{
 private:
 int x;
 public:
 bool veraendern(int);
 int aktwert(void);
};
```

Wir müssen die Methoden *veraendern* und *aktwert* explizit als *public* deklarieren, weil Methoden genau wie Attribute privat oder öffentlich sein können.

**Methodendefinition**

In der Klassendefinition befinden sich nur die Deklarationen der Methode, deswegen müssen wir noch ihre Definitionen hinzufügen: Schauen wir uns zunächst die Methode *aktwert* an:

```
int testklasse::aktwert(void)
{
  return(x);
}
```

1. Deswegen werden Methoden in C++ auch als »Elementfunktionen« bezeichnet.

Um dem Compiler anzugeben, dass dies die Definition der Methode *aktwert* der Klasse *testklasse* ist[2], wird dem Namen der Methode der Klassenname, gefolgt vom Bezugs-Operator, vorangestellt. Da sowohl die Methoden einer Klasse als auch deren Attribute den gleichen Bezugsrahmen besitzen[3], braucht beim Zugriff auf *x* innerhalb der Methode *aktwert* kein Bezugsoperator benutzt zu werden. Schauen wir uns nun die Funktion *veraendern* an:

```
bool testklasse::veraendern(int a)
{
  if((a>=0)&&(a<=10))
    {
      x=a;
      return(true);
    }
  return(false);
}
```

Da das Attribut *x* in der Klasse gekapselt ist, kann *x* von einem Benutzer der Klasse nur über die Methode *veraendern* verändert werden. Zwangsläufig muss er sich dann auch an die Beschränkungen halten, die ihm *veraendern* auferlegt. Er hat daher keine Chance, dem Attribut *x* einen Wert außerhalb des Intervalls [0,10] zuzuweisen.

Hier sehen Sie sehr schön, warum das oft vorgebrachte Argument »Wenn man ein Attribut sowieso über eine Methode verändern kann, dann könnte man das Attribut auch gleich direkt verändern« nicht gilt. Dieses Argument gilt selbst dann nicht, wenn die Methode *veraendern* folgendermaßen aussehen würde:

```
int testklasse::veraendern(int a)
{
  x=a;
}
```

Obwohl die Methode nun identisch ist mit einer direkten Zuweisung an *x*, muss ein Benutzer Veränderungen über die Methode *veraendern* vornehmen. Dies hat den Vorteil, dass Sie auch im Nachhinein noch Abfragen bezüglich der Bereichsüberschreitung von *x* implementieren können, ohne dass sich für einen Benutzer der Klasse die Schnittstelle ändert.

Wäre dagegen das Attribut *x* zuerst direkt zugänglich gewesen – also öffentlich – und erst bei der Einführung einer Bereichsprüfung privat geworden, müssten alle Benutzer der Klasse ihre Programme abändern.

Die Implementation von *testklasse* finden Sie mitsamt einer *main*-Funktion zum Testen auf der CD unter /BUCH/KAP03/BSP002.

Wir wollen an dieser Stelle einmal eine der neuen Eigenschaften von C++ anwenden.

2. Da jede Klasse ihren eigenen Bezugsrahmen besitzt, können Methoden unterschiedlicher Klassen durchaus gleiche Namen besitzen.
3. Nämlich den Bezugsrahmen der entsprechenden Klasse.

Die ursprüngliche Form der Methode *veraendern* benutzt als Funktionsparameter die Variable a. Schreiben Sie *veraendern* einmal so um, dass der Name des Funktionsparameters nicht mehr *a*, sondern *x* lautet.

Hier ist die Lösung:

```
bool testklasse::veraendern(int x)
{
  if((x>=0)&&(x<=10))
    {
      ::x=x;
      return(true);
    }
  return(false);
}
```

Der Trick besteht darin, den Scope-Operator zu verwenden. Die Verwendung des Scope-Operators ist in diesem Fall nur als Übung zu verstehen. Der Lesbarkeit wegen sollten Sie die alte Version mit *a* als Funktionsparameter beibehalten.

Wir sind nun an einem Punkt angelangt, an dem wir in der Lage sind, das im vorigen Kapitel entworfene Modell von Schweinen und Kartoffeln in C++ auszudrücken.

Bevor Sie sich nun den Klassenentwurf ansehen, sollten Sie zuerst einmal selbst versuchen, die Klassen in C++ zu formulieren. Es reicht, wenn Sie nur die Klasse entwerfen, ohne die Methoden wirklich zu implementieren.

Gut, dann vergleichen wir einmal die Ansätze. Zuerst die Klasse *Kartoffel*:

```
class Kartoffel
{
  private:
  int knollendicke;
  int knollenanzahl;
  int groesse;
  int bluetenanzahl;
  void calcgroesse(void);

  public:
  void wachsen(void);
  void bluehen(void);
  int pfluecken(void);
  void init(int, int, int, int);
};
```

**Aufteilung der Methoden in Gruppen**

Das einzig Interessante an diesem Klassenentwurf könnten die Methoden *pfluecken*, *init* und *calcgroesse* sein, denn sie kamen in unserem Modell überhaupt nicht vor. Wenn Sie sich noch einmal das Modell anschauen, werden Sie sehen, dass sich die Methode *fressen* von *Schwein* nicht direkt an den Attributen von *Kartoffel* zu schaffen macht, sondern sich an das Objekt selbst wendet. Als Lösungsansatz wurde auf eine mögliche *wirdGefressen*-Methode hingewiesen, die eine Manipulation der Kartoffel für die *fressen*-Methode des Schweins möglich macht, ohne die Datenkapselung aufzuweichen.

Wir werden hier eine allgemeiner gehaltene Methode namens *pfluecken* einführen[4], die auch noch von einer hypothetischen *Bauer*-Klasse angesprochen werden könnte.

Die Methode *init* wird benötigt, weil bei der Definition eines *Kartoffel*-Exemplars die einzelnen Attribute noch undefiniert sind. Mit *init* sind wir dann in der Lage, diese Attribute zu definieren.

Die Methode *calcgroesse* findet Verwendung bei der Größenberechnung der Kartoffelpflanze. In unserem stark vereinfachten Modell wurde die Größe von der Knollendicke und der Knollenanzahl abhängig gemacht. *calcgroesse* wurde deswegen als privat deklariert, weil es keinen Sinn macht, sie außerhalb der Klasse aufzurufen.

Die Methode *init* wird als Verwaltungsmethode und *calcgroesse* als Hilfsmethode bezeichnet. Lippman teilt die Methoden sinnvollerweise in folgende Gruppen ein (vgl. [LIPPMAN95]):

- **Verwaltungsmethoden**. Zu dieser Gruppe zählen alle Funktionen, die technische Aufgaben der Klasse, wie zum Beispiel Initialisierungen und Speicherreservierungen, erledigen. Hierzu gehören auch die Konstruktoren und Destruktoren, die wir im nächsten Abschnitt kennen lernen werden.
- **Implementierungsmethoden**. Hierzu zählen alle Methoden, die die Funktionalität der Klasse ausmachen. Die Implementierungsmethoden von Kartoffel sind *wachsen*, *bluehen* und *pfluecken*.
- **Hilfsmethoden**. Hier werden die Methoden zusammengefasst, die unterstützend zu den anderen Methoden kleinere Aufgaben bewältigen. Im Allgemeinen werden Hilfsmethoden als privat deklariert, weil sie nur im Zusammenhang mit den sie aufrufenden Methoden sinnvoll sind. In unserem Beispiel ist *calcgroesse* eine typische Hilfsmethode.
- **Zugriffsmethoden**. Als Zugriffsmethoden bezeichnet man die Methoden, die private Attribute der Klasse verändern dürfen. Sinnvollerweise sollte man die Anzahl der Zugriffsmethoden gering halten, um eine eventuelle Fehlersuche zu vereinfachen. In unserem Beispiel wurden keine besonderen Zugriffsmethoden definiert.

4. Da die Kartoffelknollen unter der Erde wachsen, ist *pfluecken* nicht unbedingt die treffendste Bezeichnung. Aber sie ist prägnant. Vorlieben für Verben wie *ausgraben* oder *ernten* können in den eigenen Lösungen natürlich gerne Verwendung finden.

Schauen wir uns nun eine mögliche Implementierung der Methoden von *Kartoffel* an:

```
void Kartoffel::init(int kd, int ka, int gr, int ba)
{
  knollendicke=kd;
  knollenanzahl=ka;
  groesse=gr;
  bluetenanzahl=ba;
}

void Kartoffel::calcgroesse(void)
{
  groesse=knollenanzahl*knollendicke;
}

void Kartoffel::bluehen(void)
{
  bluetenanzahl++;
}

void Kartoffel::wachsen(void)
{
  if(!bluetenanzahl)
    {
      cout << "Keine Blueten vorhanden!" << endl;
      return;
    }
  bluetenanzahl--;
  if(knollendicke<20)
    knollendicke++;
  knollenanzahl++;
  calcgroesse();
}

int Kartoffel::pfluecken(void)
{
  if(!knollenanzahl) return(0);

  knollenanzahl--;
  calcgroesse();
  return(knollendicke);
}
```

Als Nächstes kommt die Klasse *Schwein* an die Reihe:

```
class Schwein
{
  private:
  int groesse;
  int gewicht;
  int saettigungsgrad;

  public:
  void wachsen(void);
  void bewegen(void);
  void fressen(Kartoffel&);
  void init(int, int, int);
};
```

Auch hier findet sich wieder eine Methode namens *init*, die für die Initialisierung der Attribute zuständig ist.

Wichtig ist auch die Tatsache, dass bei der Methode *fressen* der Verweis auf *Kartoffel* nicht als Zeiger oder lokale Kopie, sondern als Referenz implementiert wurde. Dies hat den Vorteil, dass weder Speicherplatz noch Zeit damit verschwendet wird, eine lokale Kopie des Objekts anzulegen, was insbesondere bei größeren Klassen enorme Laufzeitvorteile mit sich bringt.

Verweise auf Klassen sollten – wenn möglich – immer als Referenz und nicht als Zeiger implementiert werden.

Schauen wir uns nun noch eine mögliche Implementierung der Methoden von *Schwein* an:

```
void Schwein::init(int gr, int ge, int sg)
{
  groesse=gr;
  gewicht=ge;
  saettigungsgrad=sg;
}

void Schwein::wachsen(void)
{
  if(saettigungsgrad>=8)
    {
      saettigungsgrad-=8;
      if(groesse<30)
        groesse++;
      if(gewicht<40)
        gewicht++;
    }
  else
```

```
    cout << "Schwein zu hungrig!" << endl;
}

void Schwein::bewegen(void)
{
  if(saettigungsgrad>=4)
    {
      saettigungsgrad-=4;
      return;
    }
  if(gewicht>10)
    {
      gewicht--;
      return;
    }
  cout << "Schwein hat keine Energiereserven mehr!" << endl;
}

void Schwein::fressen(Kartoffel &k)
{
  int naehrwert=k.pfluecken();
  if(!naehrwert)
    {
      cout << "Kartoffel besitzt keine Knollen!" << endl;
    }
  else
    {
      saettigungsgrad+=naehrwert;
      if(saettigungsgrad>100)
        {
          gewicht+=(saettigungsgrad-100)/2;
          saettigungsgrad=100;
        }
    }
}
```

Unter /BUCH/KAP03/BSP003.CPP finden Sie beide Klassenentwürfe sowie die Implementierung ihrer Methoden. Sie sollten dazu nun eine *main*-Funktion schreiben, mit Hilfe derer Sie ein wenig mit den Klassen spielen können. Versuchen Sie einige Exemplare* der Klassen zu erzeugen. Denken Sie daran, die Attribute vor dem ersten Benutzen mit *init* zu initialisieren.

* Exemplare sind individuelle Objekte einer Klasse. Wenn Sie zum Beispiel eine Variable *helmut* vom Typ *Schwein* definieren, dann ist das Objekt *helmut* ein Exemplar der Klasse *Schwein*.

### 3.3.1 inline

Wir haben die Möglichkeit, Methoden als *inline* zu deklarieren, bereits kennen gelernt. Eine solche *inline*-Methode soll auch bei unserer Testklasse zum Einsatz kommen, weswegen wir diese Gelegenheit direkt nutzen werden, einen Spezialfall der *inline*-Deklaration zu besprechen:

Steht die Methodendefinition unmittelbar in der Klassendefinition, so wird die Methode automatisch *inline*.

Am Beispiel unserer Testklasse sähe das so aus:

```
class testklasse
{
  private:
  int x;

  public:
  bool veraendern(int);
  int aktwert(void)
    {
      return(x);
    }
};
```

Die Methode *aktwert* wurde direkt in die Klassendefinition aufgenommen und wird somit automatisch zu einer *inline*-Funktion[5]. Für den Fall, dass eine Methode innerhalb der Klassendeklaration definiert wird, kann auf die Erwähnung des Klassennamens (testklasse::) im Methodennamen verzichtet werden, weil die Klassenzugehörigkeit der Methode eindeutig ist.

## 3.4 Konstruktoren und Destruktoren

**Motivation**

Vielleicht haben Sie sich ein wenig Gedanken über die Methode *init* gemacht. Eigentlich passt dieses Initialisieren nach dem Definieren gar nicht in das objektorientierte Modell. In der realen Welt hat ein Objekt bereits bei seiner Entstehung/Erschaffung alle Parameter, die es definieren, festgelegt. Die Reihenfolge des »zuerst Erschaffen und dann Parameter festlegen« wirkt ausgesprochen künstlich und konstruiert. Schlimmer wird es noch, wenn Sie für ein Objekt dynamisch Speicher reservieren müssen. Dann brauchen Sie nicht nur eine *init*-Methode, die den Speicher reserviert, sondern auch noch eine *destroy*-Methode, die den Speicher wieder freigibt, und zwar bevor das Objekt selbst gelöscht wird.

5. Es sei denn, es gäbe gute Gründe für den Compiler, sie nicht als *inline* zu deklarieren. Sie erinnern sich: Die Deklaration als *inline* ist für den Compiler nur eine Empfehlung, kein Zwang.

Diese Beobachtung, dass sowohl bei der Erschaffung als auch bei der Zerstörung von Objekten bestimmte dynamische Prozesse ablaufen, haben auch die Entwickler von C++ angestellt und haben eine Möglichkeit gefunden, dies elegant in den Griff zu bekommen. Und zwar gibt es für jede Klasse zwei besondere Methoden. Die eine wird automatisch bei der Erschaffung aufgerufen und heißt **Konstruktor**. Die andere wird automatisch vor der Löschung (auch »Zerstörung« genannt) des Objekts aufgerufen und heißt **Destruktor**. Um die Konstruktoren und Destruktoren an einem einfachen Beispiel zu veranschaulichen, werden wir die Klasse *testklasse* aus dem vorherigen Abschnitt um ebensolche erweitern:

```
class testklasse
{
  private:
  int x;

  public:
  int veraendern(int);
  int aktwert(void);
  testklasse(void);
  ~testklasse();
};
```

Schauen wir uns zuerst die Regel der Namensvergabe an.

Konstruktoren haben denselben Namen wie ihre Klasse. Destruktoren haben denselben Namen wie ihre Klasse zuzüglich einer vorangestellten Tilde.

Es ist wichtig zu beachten, dass weder der Konstruktor noch der Destruktor einen Rückgabewert besitzen.

Konstruktoren und Destruktoren besitzen keinen Rückgabewert.

Zudem besitzen Destruktoren auch keine Funktionsparameter.

Destruktoren besitzen keine Funktionsparameter.

Lediglich der Konstruktor kann mit Funktionsparametern versehen werden, die dann bei der Definition des Objekts übergeben werden müssen.

Schauen wir uns nun einmal den Konstruktor und den Destruktor für die Klasse *testklasse* an:

```
testklasse::testklasse(void)
{
  cout << "Objekt wurde erzeugt." << endl;
}
```

```
testklasse::~testklasse()
{
  cout << "Objekt wurde geloescht." << endl;
}
```

Konstruktor und Destruktor erledigen in diesem Fall keine sinnvollen Aufgaben. Jedoch können Sie sich unter /BUCH/KAP03/BSP004.CPP auf der CD die Verhaltensweisen einmal anschauen.

Interessant werden die Funktionsparameter des Konstruktors bei unseren Klassen *Schwein* und *Kartoffel*. Dort ersetzen sie auf elegante Weise die *init*-Methoden. Schauen wir uns einmal die neue Klassendefinition mitsamt ihren Konstruktoren an:

```
class Kartoffel
{
  private:
  int knollendicke;
  int knollenanzahl;
  int groesse;
  int bluetenanzahl;
  void calcgroesse(void);

  public:
  void wachsen(void);
  void bluehen(void);
  int pfluecken(void);
  Kartoffel(int, int, int, int);
};

Kartoffel::Kartoffel(int kd, int ka, int gr, int ba)
{
  knollendicke=kd;
  knollenanzahl=ka;
  groesse=gr;
  bluetenanzahl=ba;
}

class Schwein
{
  private:
  int groesse;
  int gewicht;
  int saettigungsgrad;

  public:
  void wachsen(void);
  void bewegen(void);
  void fressen(Kartoffel&);
```

```
  Schwein(int, int, int);
};

Schwein::Schwein(int gr, int ge, int sg)
{
  groesse=gr;
  gewicht=ge;
  saettigungsgrad=sg;
}
```

Eine Definition eines Objekts vom Typ *Schwein* könnte wie folgt aussehen:

```
Schwein s1(25,20,80);
```

Zum Experimentieren finden Sie die Klassen *Schwein* und *Kartoffel* mitsamt ihren Konstruktoren auf der CD unter /BUCH/KAP03/BSP005.CPP.

So wie Attribute und andere Methoden auch, können Konstruktoren sowohl privat als auch öffentlich sein. Allerdings sind private Konstruktoren – wie normale Methoden auch – nicht von außen benutzbar. Nur eine Methode der Klasse kann mit Hilfe eines privaten Konstruktors ein neues Exemplar erzeugen.

## 3.5 Die Elementinitialisierungsliste

Wir haben mit den Konstruktoren eine elegante Möglichkeit gefunden, die Attribute einer Klasse zu initialisieren. In den Fällen, die wir bisher kennen gelernt haben, stießen wir auch auf keine Probleme. Schauen wir uns jedoch nun als Beispiel die Klasse *Referenz* an:

```
class Referenz
{
  private:
  int &a;
  int b;

  public:
  Referenz(int&);
  void print(void);
};
```

Der Konstruktor ist so deklariert, dass ihm eine Referenz vom Typ *int* übergeben wird. Der Konstruktor selbst soll nun mit dieser Referenz das Attribut *a* initialisieren. Das Attribut *b* soll immer fest mit 3 initialisiert werden. Der Konstruktor könnte folgendermaßen aussehen:

```
Referenz::Referenz(int &r)
{
  a=r;
```

```
  b=3;
}
```

Fällt Ihnen etwas auf? Wenn Ihnen noch nichts aufgefallen sein sollte, wird Sie wohl spätestens die folgende Übung stutzig machen.

Bevor Sie weiterlesen, versuchen Sie einmal, die Klasse mitsamt ihres Konstruktors zu kompilieren. Es wird ein Fehler auftreten. Wenn Ihnen die Ursache des Fehlers nicht klar sein sollte, lesen Sie noch einmal den Abschnitt über Referenzen nach.

Der Fehler tritt deswegen auf, weil Referenzen bei ihrer Definition initialisiert werden müssen. Für die Zuweisung von *r* an *a* im Konstruktor ist es bereits zu spät. Glücklicherweise kann man von jeder Variablen explizit einen Konstruktor aufrufen, der die Initialisierung vornimmt. Damit beim Aufruf des Konstruktors von *Referenz* auch der Konstruktor von *a* aufgerufen wird, muss dieser in der **Elementinitialisierungsliste** von *Referenz* stehen.

Die Elementinitialisierungsliste folgt dem Funktionskopf und ist durch einen Doppelpunkt von ihm getrennt.

Der neue Konstruktor sieht dann folgendermaßen aus:

```
Referenz::Referenz(int &r) : a(r)
{
  b=3;
}
```

Diesen Konstruktor können Sie einwandfrei kompilieren. Da die Elementinitialisierungsliste mehrere Parameter besitzen kann, können Sie die Initialisierung von *b* ebenfalls dorthin versetzen.

Die einzelnen Parameter der Elementinitialisierungsliste werden durch Kommata voneinander getrennt.

Der endgültige Konstruktor sieht dann so aus:

```
Referenz::Referenz(int &r) : a(r),b(3)
{
}
```

Die Klasse mitsamt Konstruktor und *main*-Funktion zum Ausprobieren finden Sie auf der CD unter /BUCH/KAP03/BSP006.CPP.

## 3.6 Statische Attribute

Wenn Sie mehrere Exemplare einer Klasse erzeugen, dann hat jede Exemplar ihre eigenen Attribute. Zum Beispiel können Sie vier Objekte vom Typ *Schwein* erzeugen und jedem einzelnen Objekt ein eigenes Gewicht, eine eigene Größe etc. zuweisen.

**static**

Manchmal kann es jedoch nützlich sein, wenn bestimmte Attribute einer Klasse für alle Exemplare gleich sind, also sich alle Exemplare bestimmte Attribute teilen. Ein einfaches Beispiel für ein solches Attribut wäre zum Beispiel ein Zähler, der festhält, wie viele Exemplare einer Klasse erzeugt wurden. Das Schlüsselwort, um ein solches Attribut zu definieren, heißt **static**.

Mit *static* deklarierte Attribute existieren nur ein einziges Mal und gelten für alle Exemplare einer Klasse.

Schauen wir uns dazu einmal eine Klasse an:

```
class Counter
{
  private:
  static int anzahl;
  int wert;

  public:
  Counter(void);
  ~Counter();
  void print(void);
};
```

Die Klasse *Counter* beinhaltet ein statisches Attribut *anzahl*, welches die existierenden Exemplare zählt. Zudem kann jedes Exemplar noch einen individuell unterschiedlichen Wert aufnehmen. Der Konstruktor und der Destruktor der Klasse sehen folgendermaßen aus:

```
Counter::Counter(void)
{
  anzahl++;
}

Counter::~Counter()
{
  anzahl--;
}
```

**Initialisierung statischer Attribute**

Ein Punkt ist jedoch bis jetzt unberücksichtigt geblieben: Wie wird das statische Attribut initialisiert? Es darf weder in der Klassendefinition noch in einer Methode der Klasse initialisiert werden, weil statische Attribute nur einmal initialisiert werden dürfen. Wir müssen es daher von außen initialisieren:

```
int Counter::anzahl=0;
```

Bei der Erzeugung eines Objekts vom Typ *Counter* erhöht der Konstruktor das Attribut *anzahl* um eins. Sollte das Objekt wieder gelöscht werden, vermindert der Destruktor *anzahl* um eins. Eine *main*-Funktion zum Testen könnte so aussehen:

```
int main()
{
  Counter v1;
  v1.print();

  Counter v2[5];
  v1.print();
}
```

**Ort der Initialisierung**

Sollten Sie die Klasse in ihre Deklaration und in ihre Definition aufteilen, dann darf die Initialisierung der statischen Variablen auf keinen Fall bei der Klassendeklaration stehen, weil diese bei größeren Projekten häufiger vom Compiler bearbeitet werden könnte. Wenn die Initialisierung bei den Definitionen der Methoden steht, kann nichts passieren.

Die Klasse *Counter* mitsamt ihrem Konstruktor und ihrem Destruktor sowie die oben angeführte *main*-Funktion finden Sie in der Datei »BSP05-07.CPP«.

## 3.7 Statische Methoden

In C++ gibt es die Möglichkeit, dem Compiler mitzuteilen, dass eine Methode auch unabhängig von einem Klassen-Exemplar verwendet werden kann. Dies sind die **statischen Methoden**.

Dazu ein kleines Beispiel:

```
class Warnklasse
{
  public:
    static void Warnung(void)
      {
        cout << "Achtung!!" << endl;
      }
};
```

Die Methode *Warnung* der Klasse *Warnklasse* kann nun benutzt werden, ohne vorher ein Exemplar von *Warnklasse* erzeugen zu müssen. Und zwar wird die Methode über den Klassennamen und den Scope-Operator aufgerufen:

```
Warnklasse::Warnung();
```

Aufgrund der Tatsache, dass eine statische Methode nicht an ein Exemplar der Klasse gebunden ist, darf sie in keinster Weise auf Klassenattribute zugreifen. Auch nicht-statische Methoden der Klasse dürfen von einer statischen Methode nicht aufgerufen werden.

Verwendet werden statische Methoden meist dann, wenn bestimmte Hilfsmethoden der Klasse auch für andere Teile des Programms nutzbar sein sollen. Beispielsweise könnten Sie eine Klasse mit einem Zufallsgenerator ausstatten, der ja auch für andere Anwendungsgebiete außerhalb der Klasse eingesetzt werden kann.

Oder Sie kapseln alle Hilfsmethoden zu einem bestimmten Thema in einer Klasse. Die Klasse selbst würde dann nur als Gruppierungsmöglichkeit eingesetzt.

## 3.8 Überladen von Methoden

Mit dem Prinzip des Überladens haben wir uns bereits im Grundlagen-Kapitel vertraut gemacht. Natürlich können auch Methoden überladen werden.

**Überladen von Konstruktoren**

Sehr häufig wird das Überladen für den Konstruktor verwendet. Zum Beispiel wäre es für die Klasse *Counter* aus dem vorhergehenden Abschnitt interessant, zusätzlich zu dem parameterlosen Konstruktor noch einen weiteren Konstruktor zu implementieren, mit dem man das private Attribut *wert* initialisieren kann:

```
Counter(void);
Counter(int);
```

Das Programmieren des neuen Konstruktors können Sie als Übung selbst vornehmen.

## 3.9 this

Alle Klassen und Strukturen besitzen in C++ automatisch den Zeiger **this**. *this* ist ein C++-Schlüsselwort und zeigt immer auf das eigene Exemplar der Klasse. Dazu zuerst ein Beispiel, welches davon ausgeht, dass die Klasse *TestKlasse* ein Attribut namens *a* besitzt:

```
void TestKlasse::funktion(void)
{
  a=20;
  this->a=30;
}
```

Die erste Anweisung weist dem Attribut *a* den Wert 20 zu. Die zweite Anweisung weist *this->a* den Wert 30 zu. Da *this* ein Zeiger auf die eigene Klasse ist, wird über *this->a* das *a* des eigenen Exemplars angesprochen. Deswegen beziehen sich sowohl die erste als auch die zweite Anweisung auf das *a* von *TestKlasse*.

*this* wird hauptsächlich dann benötigt, wenn man in einer klasseneigenen Methode eine Referenz oder einen Zeiger auf das eigene Exemplar an andere Methoden oder Funktionen übergeben will.

Der *this*-Zeiger wird uns im weiteren Verlauf noch häufiger begegnen.

# 3.10 Konstante Klassen und Methoden

Wir sind nun fast am Ende des Kapitels über Klassen angekommen und werden uns noch mit der erweiterten Bedeutung von **const** beschäftigen. **const**

## 3.10.1 Konstanten und Variablen

Wir wissen bereits, wie man eine konstante *int*-Variable, also eine Konstante, definiert:

```
const int x=20;
```

Eine Konstante namens *x* wurde definiert und mit dem Wert 20 initialisiert.

Da der Wert einer Konstanten, wie der Name schon sagt, konstant ist, kann er auch nicht verändert werden. Folgende Anweisung erzeugt einen Kompilierungsfehler:

```
x=30;
```
**FALSCH!**

Man kann allerdings eine *int*-Variable definieren und ihr den Wert der Konstanten zuweisen:

```
int y;
y=x;
```

Weil *y* eine Kopie von *x* enthält, kann *y* ohne weiteres verändert werden:

```
y=40;
```

## 3.10.2 Zeiger auf Variablen

Man kann auch einen Zeiger auf den Typ *int* definieren:

```
int *ptr;
```

Man kann durch Zuweisen der Adresse von *y* über *ptr* den Wert von *y* ändern:

```
ptr=&y;
*ptr=10;
```

Man könnte auch auf die Idee kommen, dem Zeiger die Adresse von *x* zuzuweisen:

**FALSCH!**
```
ptr=&x;
```

Allerdings wird diese Anweisung bei der Kompilierung einen Fehler erzeugen. Durch die Zuweisung der Adresse der Konstanten an einen Zeiger auf den Variablentyp *int* wäre man in der Lage, den Wert von *x* zu verändern. Da *x* aber eine Konstante ist, muss dieser Manipulationsversuch vom Compiler verhindert werden.

### 3.10.3 Zeiger auf Konstanten

Man kann aber einen Zeiger auf *int*-Konstanten definieren:

```
const int *cptr;
```

Diesem kann man dann die Adresse von x zuweisen:

```
cptr=&x;
```

Nur ist man dann auch nicht in der Lage, den Wert von *x* zu ändern, weil es ja ein Zeiger auf eine *int*-Konstante ist. Deswegen erzeugt die folgende Anweisung einen Kompilierungsfehler:

**FALSCH!**
```
*cptr=50;
```

Man kann aber zwecks weiterer Verarbeitung des Wertes die Konstante über den Zeiger einer Variablen zuweisen:

```
*ptr=*cptr;
y=*cptr;
```

### 3.10.4 Konstante Zeiger

Wir haben bisher Konstanten und Zeiger auf Konstanten kennen gelernt. Es ist aber auch möglich, den Zeiger selbst als konstant zu definieren:

```
int a=4,b=14;
int *const ptrc=&a;
```

Weil der Zeiger *ptrc* konstant ist, muss er bei der Definition initialisiert werden. Denn nach der Definition kann ihm gerade wegen seiner Konstanz keine Adresse mehr zugewiesen werden.

Der Wert der Variablen *a* kann über den konstanten Zeiger *ptrc* verändert werden:

```
*ptrc=8;
```

Jedoch ist *ptrc* dazu verdammt, während seiner Lebensphase ausschließlich auf *a* zu zeigen. Folgende Zuweisung einer anderen Adresse erzeugt einen Fehler:

```
ptrc=&b;
```
**FALSCH!**

Ein konstanter Zeiger legt damit ähnliche Verhaltensweisen an den Tag wie eine Referenz. Nur dass die Syntax der Referenz eleganter ist.

Ein konstanter Zeiger auf einen konstanten Wert kann natürlich auch definiert werden:

```
const int c=20;
const int *const cptrc=&c;
```

Dem Zeiger *cptrc* kann weder eine neue Adresse zugewiesen werden, noch kann über ihn *c* verändert werden, weil *c* ebenfalls eine Konstante ist. Lediglich den Wert von *c* kann man mit *cptrc* auslesen:

```
cout << *cptrc << endl;
```

### 3.10.5 Konstante Attribute

Bei den Klassen kommen dem Schlüsselwort *const* noch andere Bedeutungen zu. Als Beispiel nehmen wir das folgende Grundgerüst:

```
class test
{
  private:
    int wert;
    const int cwert;

  public:
    test(int w) : wert(w), cwert(w) {}

};
```

Die Klasse hat einen variablen und einen konstanten *int*-Wert als Attribute.

Ein konstantes Attribut kann nur in der Elementinitialisierungsliste des Konstruktors initialisiert werden.

Folgender Konstruktor würde einen Fehler erzeugen:

```
test(int w) : wert(w) {cwert=w;}
```
**FALSCH!**

Der variable Wert könnte natürlich ohne weiteres im Anweisungsblock initialisiert werden.

Nehmen wir als erstes Anschauungsmaterial folgende öffentliche Methode von *test*:

```
int get(void) {return(wert);}
```

Wir können den von *get* zurückgelieferten Wert ganz normal Variablen und Konstanten zuweisen:

```
const int x=t.get();
int y=t.get();
```

Wir können auch eine Methode implementieren, die uns den konstanten Wert liefert:

```
const int getc(void) {return(cwert);}
```

Auch hier ist eine Zuweisung an Konstanten und Variablen möglich:

```
const int x=t.getc();
int y=t.getc();
```

Man hätte die Methode aber auch ohne konstanten Rückgabewert definieren können:

```
int getc(void) {return(cwert);}
```

Wieso ist das erlaubt? Nun, der von Funktionen zurückgelieferte Wert ist – sofern es sich nicht um einen Zeiger oder eine Referenz handelt – nur eine Kopie des Originalwertes. Deswegen trifft die Eigenschaft der Konstanz des Originalwertes nicht auf die Kopie zu. Aus diesem Grund spielt es auch keine Rolle, ob die Kopie ein konstanter oder ein variabler Wert ist.

Es ist egal, ob der zurückgelieferte Wert einer Variablen oder Konstanten zugewiesen wird, denn es wird eine Kopie von der Kopie angefertigt. Und die Kopie der Kopie muss nicht die gleichen Eigenschaften besitzen wie die ursprüngliche Kopie.

Anders sieht es aus, wenn wir Zeiger auf die Werte zurückliefern. Gehen wir zunächst von folgender Methode aus:

```
int *getptr(void) {return(&wert);}
```

Mit Hilfe dieser Methode sind folgende Zuweisungen durchaus zulässig:

```
const int *x=t.getptr();
int *y=t.getptr();
```

Allerdings ist man mit *x* in der Zugriffsvielfalt eingeschränkt, weil es sich um einen Zeiger auf Konstanten handelt. Es spielt dann keine Rolle, ob die Variable, auf die *x* zeigt, wirklich eine Konstante ist oder nicht.

Von den beiden folgenden Anweisungen wird die zweite einen Kompilierungsfehler erzeugen:

```
*y=60;
*x=70;
```

Obwohl *wert* eine Variable ist, wurde ihre Adresse einem Zeiger auf konstante *int*-Werte zugewiesen. Deswegen sind Änderungen über *x* nicht möglich.

### 3.10.6 Zeiger auf Konstanten als Rückgabewert

Als Nächstes ist die Konstante *cwert* an der Reihe. Wir schreiben dazu folgende Methode:

```
int *getcptr(void) {return(&cwert);}
```

**FALSCH!**

Diese Methode wird sich nicht einwandfrei kompilieren lassen. Das liegt daran, dass *cwert* eine Konstante ist. Wir können ihre Adresse nicht als Adresse einer Variablen zurückliefern, weil dadurch der Manipulation Tür und Tor geöffnet wäre. Wir müssen also den Rückgabewert als Adresse einer Konstanten definieren:

```
const int *getcptr(void) {return(&cwert);}
```

Aus dem gleichen Grund wird von den beiden folgenden Anweisungen die zweite einen Fehler bei der Kompilierung erzeugen:

```
const int *x=t.getcptr();
int *y=t.getcptr();
```

Wäre die zweite Anweisung fehlerfrei kompiliert worden, dann hätten wir die Konstante ändern können.

### 3.10.7 Zeiger auf Klassen

Gehen wir noch einen Schritt weiter und definieren Zeiger auf Klassen:

```
test t(123);
test *ptr=&t;
```

Wir definieren noch eine kleine Ausgabe-Funktion:

```
void print(void)
  {
    cout << wert << "/" << cwert << endl;
  }
```

Und schon können wir mit folgender Anweisung die beiden Werte ausgeben:

```
ptr->print();
```

### 3.10.8 Zeiger auf konstante Klassenobjekte

Wir können auch einen Zeiger auf ein konstantes Objekt definieren:

```
const test *cptr=&t;
```

Der Zeiger zeigt nun auf ein konstantes Klassenobjekt. Alle Werte dieses Objekts können nicht geändert werden, auch wenn es sich um ursprüngliche Variablen wie *wert* handelt. Allerdings können wir die Werte weiterhin ausgeben, da das Ausgeben keine Veränderung verursacht:

```
cptr->print();
```

Irrtum! Die vorige Anweisung wird bei der Kompilation einen Fehler verursachen. Da Methoden einer Klasse grundsätzlich auch variable Attribute verändern könnten, ist es nicht gewährleistet, dass die Konstanz des Exemplars eingehalten wird.

### 3.10.9 Konstanz wahrende Methoden

Um diesem Dilemma zu entgehen, muss eine Methode, die keine Änderungen an den Attributen der Klasse vornimmt, grundsätzlich als solche deklariert werden. Nur Methoden, die als konstanzwahrende Methoden gekennzeichnet sind, dürfen bei einer konstanten Klasse verwendet werden:

```
void print(void) const
  {
    cout << wert << "/" << cwert << endl;
  }
```

Die *print*-Methode ist nun als konstanzwahrend deklariert und kann deshalb auch von einer konstanten Exemplar verwendet werden. Es ist auch möglich, konstante Methoden von variablen Instanzen zu benutzen.

Es ist jedoch nicht möglich, eine Methode, die Änderungen an den Attributen vornimmt, als konstant zu deklarieren:

**FALSCH!**
```
void vermindern(void) const
  {wert--;}
```

Dies führt zwangsläufig zu einer Fehlermeldung.

### 3.10.10 Entfernen der Konstanz

Im Zusammenhang mit konstanten Zeigern kann eine kleine Unannehmlichkeit auftreten. Schauen Sie sich dazu einmal folgende Zeilen an:

```
int x=20;
int *ptr;
const int *cptr;

cptr=&x;
ptr=cptr;
```

Einem Zeiger auf konstante *int*-Werte (*cptr*) wird die Adresse einer *int*-Variablen (*x*) zugewiesen. Der Zugriff auf *x* über *cptr* unterliegt nun den üblichen Beschränkungen, denen Konstanten ausgesetzt sind.

Dann wird einem Zeiger auf *int*-Variablen (*ptr*) die in *cptr* gespeicherte Adresse zugewiesen. An dieser Stelle wird der Compiler einen Fehler melden. Das ist auch nachvollziehbar, denn sonst könnten wir durch diese Zuweisung die Konstanz der Variablen, auf die *cptr* zeigt, aufheben.

Betrachtet man aber die komplette Situation, so zeigt *cptr* nicht tatsächlich auf eine Konstante, sondern auf einen variablen *int*-Wert. Es wäre demnach

durchaus erlaubt, diese Variable im Nachhinein über *ptr* zu verändern. So weit denkt der Compiler allerdings nicht.

**const_cast**

Glücklicherweise gibt es in C++ die Möglichkeit, dem Compiler mitzuteilen, dass er prüfen soll, ob der Wert, auf den ein Zeiger auf Konstanten zeigt, auch wirklich eine Konstante ist. Ist es nämlich keine Konstante, dann wäre eine Zuweisung der Adresse an einen Zeiger auf variable Werte durchaus erlaubt.

Und dies geht wie folgt:

**Syntax**

```
ptr=const_cast<int*>(cptr);
```

Die durch *cptr* implizite Konstanz wird durch *const_cast* aufgehoben und in eine Adresse einer *int*-Variablen umgewandelt. Sie können nun über *ptr* die Variable *x* verändern.

Allerdings tritt bei unsachgemäßer Anwendung von const_cast ein Problem auf. Betrachten Sie dazu bitte folgende Anweisungen:

```
const int y=20;
int *ptr;
const int *cptr;

cptr=&y;
ptr=const_cast<int*>(cptr);
*ptr=40;
```

Nun zeigt *cptr* tatsächlich auf eine *int*-Konstante. Und obwohl nun eine Zuweisung an *ptr* unterbunden werden müsste, wird mit *const_cast* die Zuweisung fehlerfrei kompiliert. Selbst die Veränderung des in *y* gespeicherten Wertes über *ptr* wird anstandslos übersetzt.

In Wirklichkeit jedoch wird die Zuweisung von 40 – obwohl fehlerfrei kompiliert – im Programmlauf nicht ausgeführt. Wenn Sie hinter der Zuweisung den Wert von *y* ausgeben lassen, dann wird 20 ausgegeben.

Sie sehen also, dass sie mit *const_cast* äußerst gewissenhaft umgehen müssen. Denn solch ein Fehler, der zwar einwandfrei kompiliert, aber nicht korrekt ausgeführt wird, ist in einem komplexeren Programm schwer auszumachen.

## 3.10.11 Mutable

Häufig existieren innerhalb einer Klasse Attribute, die rein der internen Verwaltung vorbehalten sind und auf die von außen nicht zugegriffen werden kann. Bei diesen internen Attributen kann es manchmal sinnvoll sein, dass sie veränderbar sind, obwohl ein entsprechendes Exemplar als Konstante definiert wurde[6].

6. Wenn man beispielsweise mitzählen möchte, wie häufig auf eine Exemplar zugegriffen wird, dann muss die Zählvariable auch bei einer konstanten Exemplar veränderbar sein.

**mutable** Eine solche Ausnahme von der allgemeinen Konstanz definiert man mit dem Schlüsselwort **mutable**.

Ein Beispiel:

```
class Klasse
{
  private:
    mutable int x;
    int y;

  public:

    void Klasse::change(void) const
    {
      x=20;
    }
};
```

Wäre *x* nicht als *mutable* deklariert, dann würde der Compiler bei der Kompilation von *change* einen Fehler melden, denn normalerweise darf eine als *const* deklarierte Methode keine Attribute der Klasse verändern.

Aber so kann auch bei einer konstanten Exemplar von *Klasse* die *change*-Methode aufgerufen werden:

```
Klasse k;
const Klasse c=k;

c.change();
```

## 3.11 Freunde

Manchmal wäre es von Vorteil, Elemente einer Klasse zwar des Zugriffsschutzes wegen als privat zu deklarieren, bestimmten Klassen oder Funktionen aber ruhig unbegrenzten Zugriff auf die privaten Elemente zu gewähren.

Dies könnte sich beispielsweise anbieten, wenn Klassen programmiert werden, die sich inhaltlich sehr nahe stehen.

In C++ gibt es diese Möglichkeit in Form von Freunden der Klasse. Solche Freunde haben dann die gleichen Zugriffsrechte auf die Klasse wie die Klasse selbst. Das heißt, dass für Freunde sogar die privaten Elemente frei zugänglich sind. Die Deklaration eines Freundes sieht folgendermaßen aus:

**Syntax**

```
friend class Klasse;
```

Die Klasse *Klasse* ist nun ein Freund der Klasse, in der die *friend*-Deklaration stand.

Freunde einer Klasse haben die gleichen Zugriffsrechte wie die Methoden der Klasse selbst.

Es ist auch möglich, nur einzelne Funktionen als Freund zu deklarieren. Diese Funktion muss keine Methode sein. Jede Funktion, ob sie nun einer Klasse angehört oder nicht, kann von einer Klasse als Freund deklariert werden. Die Schreibweise sieht dann folgendermaßen aus:

```
friend int max(int, int);
```

Die Funktion *max*, die keiner Klasse angehört, ist nun ein Freund der Klasse, die die *friend*-Deklaration beinhaltet.

Die *friend*-Deklaration kann an jeder beliebigen Stelle innerhalb der Klasse stehen. Es empfiehlt sich jedoch, sie an den Anfang zu schreiben, um auch rein optisch keine Zugehörigkeit zu den *private*- oder *public*-Bezugsrahmen nahe zu legen.

*friend*-Deklarationen sollten immer am Anfang der Klassendefinition stehen.

Für eine beliebige Klasse sähe das so aus:

```
class Testklasse
{
  friend class FreundKlasse;
  private:
...
  public:
...
};
```

## 3.12 Klassendiagramme

Wir haben bisher einige Grundlagen der OOP kennen gelernt und davon auch schon Bereiche in C++ umgesetzt. Um Klassen und deren Beziehungen untereinander sprachen-unabhängig[7] darstellen zu können, bieten sich Diagramme an. Eine Möglichkeit, Klassen darzustellen, haben Sie ja bereits im vorigen Kapitel gesehen.

Nun gibt es aber bei der grafischen Darstellung so viele Variationen, wie es Professoren und Autoren gibt. Um die grafische Darstellung zu dem zu machen, was sie eigentlich sein sollte, nämlich sprachen-unabhängig und für Leute aus den unterschiedlichsten IT-Bereichen verständlich, musste eine Norm her.

7. Gemeint sind hier die Computer-Sprachen.

**Literatur**

Diese Norm scheint in der UML[8] gefunden. Dieses Buch kann nicht auf alle Aspekte der UML eingehen, dafür sind Bücher wie [OESTEREICH98] oder [GRÄSSLE00] besser geeignet.

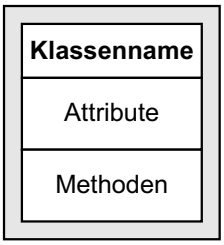
Aber wir wollen hier zumindest so weit in die UML einsteigen, dass wir die Dinge, die wir in C++ leisten, auch UML-gerecht darstellen können.

In diesem Kapitel haben wir Klassen und deren Bestandteile besprochen, sodass wir uns zunächst mit den Klassendiagramm der UML beschäftigen werden.

**Darstellung einer KLasse**

Im Klassendiagramm wird eine Klasse als rechteckiger, in drei waagerechte Bereiche unterteilter Kasten dargestellt. Der obere Bereich beinhaltet den Klassennamen, der mittlere die Attribute und der untere die Methoden, in der UML-Sprache auch Operationen genannt. Abbildung 3.1 zeigt eine allgemeine Darstellung des Sachverhalts.

*Abbildung 3.1: Die Klasse im Klassendiagramm*




Um die genaue Schreibweise der Attribute und Methoden zu besprechen, wollen wir die folgende, sinnlose Beispielklasse in einem Klassendiagramm darstellen:

```
class Beispiel1 {
  public:
    int attributo;
    bool methodeo(int wert);

  private:
    long attributp;
    long methodep(char zeichen);
};
```

**Attribut**

In dieser Klasse besteht ein Attribut aus einem Namen und einem Datentyp. Der Aufbau in der UML sieht so aus:

```
Attributname : Datentyp
```

**Zugriffsrecht**

Ob es sich um ein privates oder öffentliches Attribut handelt, geben Sie mit einem + (für öffentlich) oder einem – (für privat) vor dem Namen des Attributs an. Das Attribut *attributo* wird damit folgendermaßen formuliert:

```
+attributo : int
```

8. UML als Abkürzung für »Unified Modelling Languange«, basierend auf Booch, Rumbaugh und Jacobson, auch »die drei Amigos« genannt [BOOCH99].

**Methoden**

Methoden haben folgenden Aufbau:

```
Methodenname(Parameterliste) : Rückgabetyp
```

**Parameterliste**

Innerhalb der Parameterliste werden die einzelnen Parameter durch Kommata getrennt. Ein Parameter für sich hat folgende Struktur:

```
Richtung Parametername : Parametertyp
```

Dabei gibt die Richtung an, ob der Parameter Informationen an die Methode übergibt oder Informationen von der Methode zurückliefert.

Sollte der Parameter zur Übergabe von Daten an die Methode verwendet werden, dann heißt die Richtung »in«. Da dies der Normalfall ist, kann im Falle von »in« die Richtungsangabe weggelassen werden. Liefert die Methode über den Parameter Daten zurück (z.B. über eine Referenz oder einen Zeiger), dann wird als Richtung »out« angegeben. Wird der Parameter bidirektional genutzt, geben Sie als Richtung »inout« an.

Mit diesen Informationen lässt sich unsere Beispielklasse bereits in einem Klassendiagramm darstellen. Abbildung 3.2 zeigt die Umsetzung.

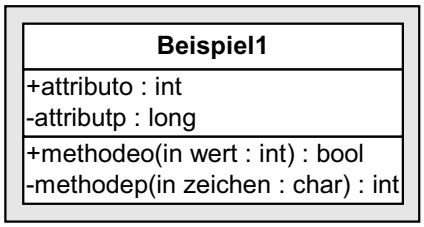



*Abbildung 3.2: Eine Beispielklasse im Klassendiagramm*

**Start- und Standardwerte**

Attribute einer Klasse können Startwerte[9] und Methoden-Parameter Standardwerte besitzen. Möchten Sie dies im Klassendiagramm darstellen, dann schreiben Sie es einfach mit einem Gleichheitszeichen hinter den Datentyp des betroffenen Attributs. Abbildung 3.3 zeigt die Beispielklasse *Beispiel1*, deren Attribut *attributo* nun den Startwert 42 besitzt. Der Parameter *wert* der Methode *methodeo* hat als Standardargument den Wert 36.

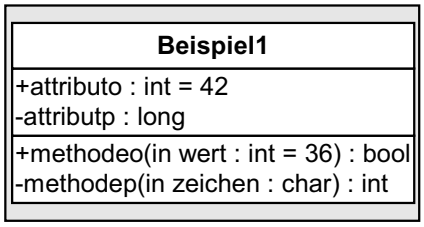
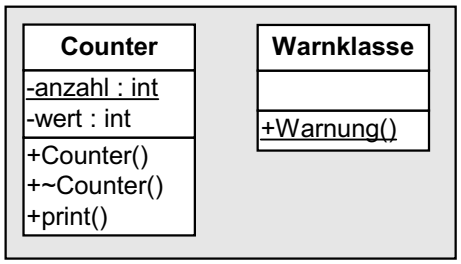



*Abbildung 3.3: Start- und Standardwerte*

9. Startwerte wären beispielsweise die Werte, die ein Attribut bei der Initialisierung der Klasse – also durch den Konstruktor – zugewiesen bekommt.

Statische Methoden und Attribute werden im Klassendiagramm unterstrichen dargestellt. Abbildung 3.4 zeigt die weiter oben im Kapitel besprochenen Klassen *Counter* (mit statischem Attribut) und *Warnklasse* (mit statischer Methode.)

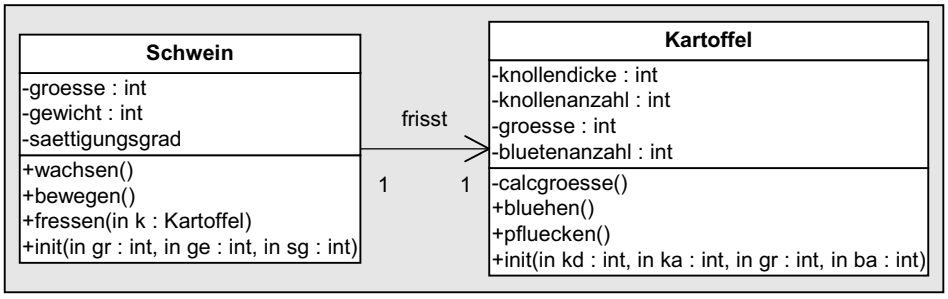
*Abbildung 3.4: Die Klassen Counter und Warnklasse*




Setzen sie als kleine Übung einmal die beiden Klassen *Schwein* und *Kartoffel* in ein UML-Klassendiagramm um.

Das entsprechende Diagramm ist mit einer kleinen Neuerung in Abbildung 3.5 zu sehen.

*Abbildung 3.5: Schweine und Kartoffeln in der UML*




**Assoziation**

Die Klassen *Schwein* und *Kartoffel* standen in unserer anfänglichen Überlegung in einer Beziehung, nämlich dass ein Schwein eine Kartoffel fressen kann. Eine solche Beziehung nennt man in der UML **Assoziation**. Da diese Assoziation nur in eine Richtung geht, wird sie als **gerichtete Assoziation** bezeichnet.

Die Zahlen geben die so genannte Multiplizität an. In diesem Fall bedeutet es, dass genau ein Schwein genau eine Kartoffel frisst. Natürlich kann ein Schwein hintereinander mehrere Kartoffeln fressen, aber der Funktion *fressen* wird immer nur eine Kartoffel und kein Feld von Kartoffeln übergeben.

**Exemplare**

Exemplare der Klasse werden ähnlich wie die Klasse selbst dargestellt. Abbildung 3.6 verdeutlicht dies.

**Instance of**

Hinter dem Namen des Exemplars steht durch einen Doppelpunkt getrennt der Klassenname, der das Exemplar angehört. Diese Zugehörigkeit wird auch durch den Stereotyp »instance« bzw. »instance of« ausgedrückt.

Der untere Abschnitt, der die Werte der Attribute darstellt, ist optional.

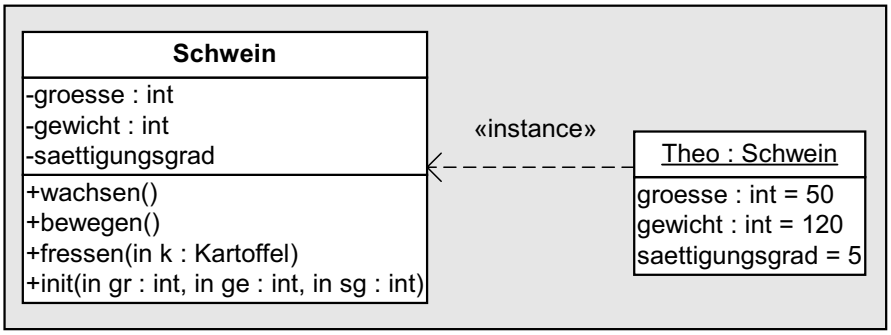



*Abbildung 3.6: Ein Exemplar in der UML*

## 3.13 Kontrollfragen

1. Zählen Sie Unterschiede zwischen *struct* und *class* auf.
2. Zählen Sie die Vorteile privater Attribute auf.
3. Wo sollte die Initialisierung eines statischen Attributes im Programm stehen und wo nicht?
4. Welche Bedingung muss berücksichtigt werden, wenn Funktionen überladen werden sollen, und welche Unterschiede reichen für ein Überladen nicht aus?
5. Dürfen Konstruktoren und Destruktoren überladen werden?
6. Welchen Sinn sehen Sie in privaten Methoden?
7. Der Konstruktor einer Klasse wird bei der Erzeugung einer Exemplar aufgerufen. Weil ihm dabei eventuelle Parameter übergeben werden, haben wir ihn als öffentlich deklariert. Machen auch private Konstruktoren Sinn?
8. Besitzen Strukturen auch den impliziten Zeiger *this*?