Übungsblatt 3

Ralph Gauges Ursula Rost Katja Wegner

07.11.2007

Aufgabe 1:

Nehmen Sie die Funktionen die Sie für die Aufgaben vom 31.10. geschrieben haben und teilen Sie diese auf in eine Header Datei mit den Deklarationen und eine Source Datei mit der Implementierung. Die Funktionen lassen sich dann einfach in weiteren Projekten verwenden.

```
1 #ifndef FUNKTIONEN_H__
2 #define FUNKTIONEN_H__
3
4 bool istEineZahl(char zeichen1,char zeichen2);
5 unsigned int umwandelnInZahl(char zeichen1,char zeichen2);
6 double fakultaet(unsigned int n);
7 unsigned int wertVonBit(unsigned int bitNummer);
8 void binaerzahlAusgeben(unsigned int n);
9 #endif // FUNKTIONEN_H__
```

So in etwa sollte die Header Datei aussehen. Wichtig sind die Präprozessordirektiven (**ifndef**, **define** und **endif**) am Anfang und am Ende. Es ist auch immer ganz gut sich zu den **endif** Direktiven das Symbol, auf das sie sich beziehen, dazuzuschreiben, da es sonst bei vielen verschachtelten **ifdef** oder **ifndef** Direktiven schnell unübersichtlich wird.

Die Implementierung der Funktionen lasse ich hier aus, da sich da nichts geändert hat.

Aufgabe 2:

Schreiben Sie eine Klasse Bruch, welche eine rationale Zahl (Bruchzahl) abbildet. Der Zähler und der Nenner sollen vom Typ **unsigned int** sein, das Vorzeichen des Bruchs soll in einem Attribut vom Typ **bool** gespeichert werden. Die Klasse soll Methoden haben um zwei Bruchzahlen zu addieren, zu subtrahieren, zu multiplizieren und zu dividieren. Die Funktionen sollen jeweils zwei Argumente vom Typ *Bruch* entgegennehmen und das Ergebnis als Wert vom Type *Bruch* zurückliefern. Die jeweiligen Ergebnisse müssen keine gekürzten Brüche darstellen. Die Klasse soll in eine Header Datei für die Deklarationen und eine Source Datei für die Definitionen aufgeteilt werden. Gleiches gilt für eventuell zusätzlich benötigte Funktionsdeklarationen und -definitionen. Die *main* Funktion soll vom Benutzer vier separate, maximal zweistellige Zahlen entgegen nehmen, wobei die ersten beiden Zähler und Nenner der ersten Bruchs sind und die letzten beiden Zähler und Nenner des zweiten Bruchs. Damit soll dann jeweils eine Instanz der Klasse Bruch erzeugt werden und das Ergebnis der Addition, Subtraktion, Multiplikation und Division der beiden Brüche ausgegeben werden. (Die Eingabe der Zahl durch den Benutzer soll das Programm auf Korrektheit überprüfen und die Eingabe in Zahlenwerte umwandeln mit denen die Brüche initialisiert werden. Bei fehlerhafter Eingabe ist eine entsprechende Fehlermeldung auszugeben. Achten Sie insbesondere auch darauf, dass der Benutzer sinvolle Werte für Zähler und Nenner eingibt.)

---------- Bruch.h ----------

```
1  #ifndef BRUCH_H__
2  #define BRUCH_H__
3
4  class Bruch
5  {
6      private:
7          bool negative;
8          unsigned int zaehler;
9          unsigned int nenner;
10
11     public:
12         Bruch();
13         Bruch(bool neg,unsigned int z,unsigned int n);
14         unsigned int getZaehler();
15         void setZaehler(unsigned int z);
16         unsigned int getNenner();
```

```
17          void setNenner(unsigned int n);
18          bool isNegative();
19          void setNegative(bool neg);
20          Bruch add(Bruch b1,Bruch b2);
21          Bruch subtract(Bruch b1,Bruch b2);
22          Bruch divide(Bruch b1,Bruch b2);
23          Bruch multiply(Bruch b1,Bruch b2);
24  };
25
26  #endif // BRUCH_H__
```

```
-------------------------- ----------
---------------- Bruch.cpp ----------
```

```
1  #include "Bruch.h"
2
3  Bruch::Bruch():negative(false),zaehler(1),nenner(1){}
4
5  Bruch::Bruch(bool neg,unsigned int z,unsigned int n)
6                :negative(neg),zaehler(z),nenner(n){}
7
8  unsigned int Bruch::getZaehler(){return zaehler;}
9
10 void Bruch::setZaehler(unsigned int z){zaehler=z;}
11
12 unsigned int Bruch::getNenner(){return nenner;}
13
14 void Bruch::setNenner(unsigned int n){nenner=n;}
15
16 bool Bruch::isNegative(){return negative;}
17
18 void Bruch::setNegative(bool neg){negative=neg;}
19
20 Bruch Bruch::multiply(Bruch b1,Bruch b2)
21 {
22     Bruch result;
23     if(b1.isNegative())
24     {
25         if(b2.isNegative())
26         {
27             result.setNegative(false);
28         }
29         else
30         {
31             result.setNegative(true);
32         }
```

```
33      }
34      else
35      {
36          if(b2.isNegative())
37          {
38              result.setNegative(true);
39          }
40          else
41          {
42              result.setNegative(false);
43          }
44      }
45      result.setZaehler(b1.getZaehler()*b2.getZaehler());
46      result.setNenner(b1.getNenner()*b2.getNenner());
47      return result;
48  }
49
50  Bruch Bruch::divide(Bruch b1,Bruch b2)
51  {
52      // is dasselbe wie b1 multipliziert mit dem Kehrwert von b2
53      Bruch result;
54      if(b2.getZaehler()!=0)
55      {
56        Bruch temp(b2.isNegative(),b2.getNenner(),b2.getZaehler());
57        result=multiply(b1,temp);
58      }
59      else
60      {
61          std::cout << "Fehler. Durch 0 dividieren ist nicht erlaubt.\n";
62      }
63      return result;
64  }
65
66
67  Bruch Bruch::add(Bruch b1,Bruch b2)
68  {
69      Bruch result;
70      if(b1.isNegative()==true)
71      {
72          if(b2.isNegative()==true)
73          {
74              result.setNegative(true);
75              result.setNenner(b1.getNenner()*b2.getNenner());
76              result.setZaehler(b1.getZaehler()*b2.getNenner()
77                                +b2.getZaehler()*b1.getNenner());
```

```
78          }
79          else
80          {
81              result=subtract(b2,b1);
82          }
83      }
84      else
85      {
86          if(b2.isNegative()==true)
87          {
88              result=subtract(b1,b2);
89          }
90          else
91          {
92              result.setNenner(b1.getNenner()*b2.getNenner());
93              result.setZaehler(b1.getZaehler()*b2.getNenner()
94                                +b2.getZaehler()*b1.getNenner());
95          }
96      }
97      return result;
98  }
99
100 Bruch Bruch::subtract(Bruch b1,Bruch b2)
101 {
102     Bruch result;
103     if(b1.isNegative()==true)
104     {
105         if(b2.isNegative()==true)
106         {
107             Bruch temp(false,b2.getZaehler(),b2.getNenner());
108             result=subtract(temp,b1);
109         }
110         else
111         {
112             Bruch temp1(false,b1.getZaehler(),b1.getNenner());
113             Bruch temp2(false,b2.getZaehler(),b2.getNenner());
114             result=add(b1,b2);
115             result.setNegative(true);
116         }
117     }
118     else
119     {
120         if(b2.isNegative()==true)
121         {
122             Bruch temp(false,b2.getZaehler(),b2.getNenner());
```

```
123             result=add(b1,temp);
124         }
125         else
126         {
127             result.setNenner(b1.getNenner()*b2.getNenner());
128             int temp=b1.getZaehler()*b2.getNenner();
129             temp-=b2.getZaehler()*b1.getNenner();
130             if(temp<0)
131             {
132                 result.setNegative(true);
133                 result.setZaehler(-1*temp);
134             }
135             else
136             {
137                 result.setNegative(false);
138                 result.setZaehler(temp);
139             }
140         }
141     }
142     return result;
143 }
```

Aufgabe 3:

Schreiben Sie eine Klasse Viereck. Die Klasse soll als Attribute vier 2D Koordinaten speichern, d.h. jeweils ein x und ein y Wert vom Typ **float** und ein Attribut, ebenfalls vom Typ **float**, welches den Umfang speichern soll. Die Klasse soll ferner über folgende Methoden verfügen:

init initialisiert alle Koordinaten und den Umfang mit dem Wert $-1.0f$.

berechneUmfang berechnet den Umfang des Vierecks und speichert das Ergebnis in dem entsprechenden Attribut

berechneLaenge nimmt als Argumente 2 Koordinaten (4 **float** Werte) und berechnet die Länge der Strecke zwischen den Koordinaten und gibt das Ergebnis als **float** Wert zurück

getUmfang liefert den Wert des Attributes zurück, falls dieses noch nicht berechnet wurde, soll vorher *berechneUmfang* aufgerufen werden

Standard Konstruktor ruft die init Methode auf

Konstruktor mit 8 float Argumenten setzt die Koordinaten auf die angegebenen Werte und berechnet den Umfang

sonstiges Methoden zum Setzen und Auslesen der 4 Koordinaten

Schreiben Sie nun eine Klasse Rechteck, welche von Viereck abgeleitet ist und drei zusätzlichen Attribute für die Breite, die Höhe und die Fläche hat. Die Attribute sollen ebenfalls in einer Methode namens *init* mit dem Wert $-1.0f$ initialisiert werden. Der Umfang und die Fläche sollen anschliessend im Konstruktor berechnet werden. Zum Berechnen der Werte kann man die geerbte Methode *berechneLaenge* verwenden. Ausserdem soll die Klasse über eine Methode verfügen, welche die Fläche des Rechtecks berechnet und im entsprechenden Attribut speichert. Die Breite, Höhe und Fläche sollen über Methoden der Klasse auslesbar sein. Es soll wiederum einen Standard Konstruktor geben, welcher *init* aufruft und einen Konstruktor, der 4 **float** Werte übernimmt und die Position, sowie die Breite und Höhe setzt.
Von der Klasse Rechteck soll nun noch eine Klasse Quadrat abgeleitet werden, welche die Methoden zur Berechnung der Fläche und des Umfangs, passend für ein Quadrat, neu implementiert. Ausserdem soll es wieder einen Standard Konstruktor geben und einen Konstruktor, welcher die Position und die Breite als 3 **float** Werte übernimmt.
Die Methoden zum berechnen sollen von ausserhalb der Klasse nicht benutzt werden können. Dasselbe gilt natürlich für die Attribute. Zum Lösen dieser Aufgabe benötigen Sie einige mathematische Funktionen, z.B. um die Quadratwurzel aus eine Zahl zu ziehen. Dazu müssen sie die Datei math.h einbinden, dort gibt es die Funktion *sqrt*. Diese nimmt ein Argument vom Typ **float** und liefert als Ergebnis auch einen **float** Wert zurück.

`------------ Viereck.h ---------`

```
1 #ifndef VIERECK_H__
2 #define VIERECK_H__
3
4 class Viereck
5 {
6     protected:
7         float x1,y1;
8         float x2,y2;
9         float x3,y3;
10        float x4,y4;
```

```
11          float umfang;
12          void berechneUmfang();
13          float berechneLaenge(float x1,float y1,float x2,float y2);
14
15      public:
16          Viereck();
17          Viereck(float _x1,float _y1,float _x2,float _y2,float _x3,float _y3,float _x
18          void init();
19          float getUmfang(){return umfang;};
20
21          float getX1(){return x1;}
22          float getX2(){return x2;}
23          float getX3(){return x3;}
24          float getX4(){return x4;}
25          float getY1(){return y1;}
26          float getY2(){return y2;}
27          float getY3(){return y3;}
28          float getY4(){return y4;}
29
30          void setX1(float v);
31          void setX2(float v);
32          void setX3(float v);
33          void setX4(float v);
34          void setY1(float v);
35          void setY2(float v);
36          void setY3(float v);
37          void setY4(float v);
38
39  };
40
41  #endif // VIERECK_H__
```

Zeile 6-13: Deklariert die Attribute der Klasse Viereck und die Methoden *berechneUmfang* und *berechneLaenge* als **protected**, d.h. Nur Objekte dieser Klasse oder Objecte von Klassen die **public** von dieser Klasse abgeleitet sind können auf die Attribute und Methoden zugreifen.

Zeile 16-37 Deklariert die Methoden auf die Benutzer und andere Objekte zugreifen dürfen als **public**. Bei kurzen Methoden, d.h. solche die nur ein Attribut zurückliefern, werden oft direkt im Header implementiert. Hier wird dies bei den ganzen *get*-Methoden gemacht.

---------- Viereck.cpp ---------

```
#include "Viereck.h"
#include <math.h>

Viereck::Viereck()
{
    init();
}

Viereck::Viereck(float _x1,float _y1,
                 float _x2,float _y2,
                 float _x3,float _y3,
                 float _x4,float _y4)
  :x1(_x1),y1(_y1),x2(_x2),y2(_y2),x3(_x3),y3(_y3),x4(_x4),y4(_y4)
{
    berechneUmfang();
}

void Viereck::init()
{
    x1=-1.0f;
    x2=-1.0f;
    x3=-1.0f;
    x4=-1.0f;
    y1=-1.0f;
    y2=-1.0f;
    y3=-1.0f;
    y4=-1.0f;
    umfang=-1.0f;
}

void Viereck::berechneUmfang()
{
    umfang=berechneLaenge(x1,y1,x2,y2);
    umfang+=berechneLaenge(x2,y2,x3,y3);
    umfang+=berechneLaenge(x3,y3,x4,y4);
    umfang+=berechneLaenge(x4,y4,x1,y1);
}

float Viereck::berechneLaenge(float x1,float y1,float x2,float y2)
{
    float deltaX=x2-x1;
    float deltaY=y2-y1;
    float laenge=sqrt(deltaX*deltaX+deltaY*deltaY);
    return laenge;
}
```

```
46
47 void Viereck::setX1(float v)
48 {
49     x1=v;
50     berechneUmfang();
51 }
52
53 void Viereck::setX2(float v)
54 {
55     x2=v;
56     berechneUmfang();
57 }
58
59 void Viereck::setX3(float v)
60 {
61     x3=v;
62     berechneUmfang();
63 }
64
65 void Viereck::setX4(float v)
66 {
67     x4=v;
68     berechneUmfang();
69 }
70
71 void Viereck::setY1(float v)
72 {
73     y1=v;
74     berechneUmfang();
75 }
76
77 void Viereck::setY2(float v)
78 {
79     y2=v;
80     berechneUmfang();
81 }
82
83 void Viereck::setY3(float v)
84 {
85     y3=v;
86     berechneUmfang();
87 }
88
89 void Viereck::setY4(float v)
90 {
```

```
91      y4=v;
92      berechneUmfang();
93  }
```

Zeile 2 Bindet der Systemheader *math.h* ein. Dieser wird für die Funktion *sqrt* zur Berechnung der Quadratwurzel benötigt.

Zeile 4-7 Standardkonstruktor der alle Attribute über einen Aufruf der *init* Methode initialisiert.

Zeile 9-16 Konstruktor mit 8 float Argumenten. Die einzelen Attribute der Klasse werden aus den Argumenten in der Initialisierungsliste initialisiert. Im Funktionskörper des Konstruktors wird dann der Umfang berechnet.

Zeile 18-29 *init* Methode, welche alle Attribute mit dem Wert -1.0f initialisiert.

Zeile 31-37 Die Methode *berechneUmfang* berechnet den Umfang als Summe der vier Teilstrecken zwischen den vier Koordinatenpunkten. Die Länge der Teilstücke wird mit Hilfe der Methode *berechneLaenge* ermittelt.

Zeile 39-45 Berechnet die Länge einer Strecke, welche durch zwei Koordinatenpunkte (x1,y1) und (x2,y2) festgelegt wird. Die Koordinaten werden in Form von 4 **float** Werten der Methode als Argumente übergeben.

Zeile 47-93 Die *set*-Methoden dienen zum Setzen der einzelnen Attribute. Nachdem eine Koordinate geändert wurde muss natürlich der Umfang neu berechnet werden.

---------- Rechteck.h --------

```
1  #ifndef RECHTECK_H__
2  #define RECHTECK_H__
3
4  #include "Viereck.h"
5
6  class Rechteck:public Viereck
7  {
8      protected:
9          float breite;
```

```
10          float hoehe;
11          float flaeche;
12          void berechneFlaeche();
13          void berechneUmfang();
14
15      public:
16          Rechteck();
17          Rechteck(float _x, float _y, float _breite, float _hoehe);
18          void init();
19          float getFlaeche(){return flaeche;}
20          float getHoehe(){return hoehe;}
21          float getBreite(){return breite;}
22          void setHoehe(float _hoehe);
23          void setBreite(float _breite);
24  };
25
26  #endif // RECHTECK_H__
```

Zeile 4: Bindet den Haeder von Viereck ein. Dies ist nötig, da Rechteck von Viereck abgeleitet werden soll und Rechteck wissen muss, was es erbt.

Zeile 6:Rechteck wird **public** von Viereck abgeleitet, d.h. Rechteck erbt alle Methoden und Attribute von Viereck, die dort entweder **public** oder **protected** deklariert wurden. Konstruktoren werden nicht vererbt!!!.

Zeile 8-13: Deklariert die neuen Attribute und eineige weitere Methoden mit **protected** Zugriffsrechten.

Zeile 15-23 Deklariert die Konstruktoren und einige weitere Methoden als **public**. Die Methode *init* wurde zwar von Viereck geerbt, da die neue **init** Methode etwas anderes machen soll, wird sie hier neu deklariert (überschrieben). Wir werden später noch lernen, wie man die ursprüngliche *init* Methode von *Viereck* von der neuen *init* Methode aus aufrufen kann.

```
---------- Rechteck.cpp --------
1  #include "Rechteck.h"
2  #include <cmath>
3
4  Rechteck::Rechteck()
5  {
6      init();
```

```
7  }
8
9  Rechteck::Rechteck(float _x,float _y,float _breite,float _hoehe):x1(_x),y1(_y),breit
10 {
11     berechneUmfang();
12     berechneFlaeche();
13 }
14
15 void Rechteck::init()
16 {
17     x1=-1.0f;
18     x2=-1.0f;
19     x3=-1.0f;
20     x4=-1.0f;
21     y1=-1.0f;
22     y2=-1.0f;
23     y3=-1.0f;
24     y4=-1.0f;
25     umfang=-1.0f;
26     breite=-1.0f;
27     hoehe=-1.0f;
28     flaeche=-1.0f;
29 }
30
31 void Rechteck::berechneFlaeche()
32 {
33     flaeche=std::abs(breite*hoehe);
34 }
35
36 void Rechteck::berechneUmfang()
37 {
38     umfang=std::abs(2*breite)+std::abs(2*hoehe);
39 }
40
41 void Rechteck::setHoehe(float _hoehe)
42 {
43     hoehe=_hoehe;
44     berechneUmfang();
45     berechneFlaeche();
46 }
47
48 void Rechteck::setBreite(float _breite)
49 {
50     breite=_breite;
51     berechneUmfang();
```

```
52      berechneFlaeche();
53  }
```

Zeile 2: Bindet den Header *cmath* ein. Im Grunde ist dies dasselbe wie *math.h*, die Funktionen werden aber in den Namensraum *std* gepackt. Man muss den ganzen Funktionsaufrufen also ein *std::* voranstellen.

Zeile 4-7: Standardkonstruktor welcher die Attribute über die neue *init* Methode initialisiert.

Zeile 9-13: Konstruktor mit 4 **float** Werten setzt einige Attribute über die Initialisierungsliste und berechnet dann im Funktionskörper den Umfang und die Fläche. Der Umfang wird dabei mit der neuen *berechneUmfang* Methode berechnet. Man sollte hier noch beachten, dass dieser Konstruktor einige der geerbten Attribute nicht initialisiert, d.h. die alte *berechneUmfang* Methode würde ein falsches Ergebnis liefern. Es wäre eigentlich besser, die restlichen Attribute würden aus den gegebenen berechnet und initialisiert.

Zeile 15-29: Die neue *init* Methode initialisiert die geerbten und die neuen Attribute mit -1.0f.

Zeile 31-53: Die Methoden *setBreite* und *setHoehe* setzen die entsprechenden Attribute, dabei muss der Umfang und die Fläche anschliessend neu berechnet werden. Auch hier wäre es besser, die Methoden würden die geerbten Attribute, die durch das setzen der Werte beeinflusst werden, aktualisieren. Die Methode *berechneFlaeche* ist neu und berechnet die Fläche des Rechtecks und setzt das entsprechende Attribut. Die Methode *berechneUmfang* überschreibt die entsprechende Methode von Viereck und implementiert sie neu.

---------- Quadrat.h --------

```
1  #ifndef QUADRAT_H__
2  #define QUADRAT_H__
3
4  #include "Rechteck.h"
5
6  class Quadrat:public Rechteck
7  {
8      public:
```

```
9           Quadrat();
10          Quadrat(float _x, float _y, float _seitenlaenge);
11      protected:
12          void berechneUmfang();
13          void berechneFlaeche();
14  };
15
16  #endif // QUADRAT_H_
```

Zeile 4-13 Ähnlich wie Rechteck wird hier die Klasse Quadrat deklariert, welche ihererseits von Rechteck abgeleietet ist. Die Kasse deklariert ihre eigenen Konstruktoren, da diese nicht vererbt werden. Der Konstruktor mir Argumenten nimmt diesmal drei Argumente, da bei einem Quadrat die Höhe und Breite gleich sind. Desweiteren werden die Methoden *berechneUmfang* und *berechneFlaeche* neu deklariert.

```
---------- Quadrat.cpp --------
1  #include "Quadrat.h"
2
3  Quadrat::Quadrat()
4  {
5      init();
6  }
7
8  Quadrat::Quadrat(float _x, float _y, float _seitenlaenge)
9  {
10     x1=_x;
11     y1=_y;
12     breite=_seitenlaenge;
13     hoehe=breite;
14     berechneUmfang();
15     berechneFlaeche();
16 }
17
18 void Quadrat::berechneUmfang()
19 {
20     umfang=4*breite;
21 }
22
23 void Quadrat::berechneFlaeche()
24 {
25     flaeche=breite*breite;
26 }
```

Zeile 3-6: Standardkonstruktor welcher die Attribute über die von Rechteck geerbte Methode *init* initialisiert. Dies reicht aus, da Quadrat gegenüber Rechteck keine neuen Attribute deklariert.

Zeile 8-16: Der Konstruktor nimmt drei **float** Argumente entgegen und setzt im Funktionskörper die drei entsprechenden Attribute. Anschliessend werden der Umfang und die Fläche des Quadrats neu berechnet. Auch hier sollte man beachten, dass die Methode nicht ganz sauber geschrieben ist, da sie einige der vom Viereck geerbten Attribute nicht aktualisiert.

Zeile 18-26 Implementiert die Methoden *berechneUmfang* und *berechneFlaeche* neu.

Aufgabe 4:

Stellen Sie Sich vor, Sie werden vor das Problem gestellt, eine Medienverwaltung für eine Bibliothek zu schreiben. Die Bibliothek verfügt über folgende Medientypen: Bücher, Zeitschriften, Musikkasetten, Musik CDs, Musik DVDs, Schallplatten, Video DVDs und Videokasetten. Finden Sie eine geeignete Klassenhierarchie und überlegen Sie Sich welche Attribute und Methoden für die einzelnen Klassen sinnvoll sein könnten. (Tipp: Überlegen Sie welche Gemeinsamkeiten und Unterschiede die einzelnen Medientypen haben. Z.B. Kann man vielleicht alle diese Dinge ausleihen und die Bibliothek muss vermerken, wer was wann ausgeliehen hat, auf der anderen Seite kann man sicherlich nur für Bücher und Zeitschriften eine Seitenzahl sinnvoll speichern.) Versuchen Sie anschliessend diese Klassenhierarchy zu implementieren.
Diese Aufgabe lässt sich auf viele unterschiedliche Arten lösen und die Hierarchie der einzelnen Klassen und deren Attribute werden sicherlich bei jedem etwas anders aussehen. Abbildung 1 beschreibt eine Möglichkeit einer Klassenhierarchie. Man könnte sicherlich noch mehr Klassen hinzufügen wie z.B. Computerspiele und die einzelnen Klassen sind durch die angegebenen Attribute sicherlich nicht vollständig beschrieben. Was aus dem Diagramm jedoch hervorgehen sollte ist, dass es eine gemeinsame Klasse Medium gibt von der alle anderen Klassen abgeleitet sind. Diese Klasse enthält die Attribute, die für jedes Medium wichtig sind. In diesem Fall speichert die Basisklasse z.B. wer das Objekt, wann ausgeliehen hat und wann er es zurückgeben sollte. Von dieser Basisklasse Medium sind drei weitere Klassen abgeleitet, nämlich

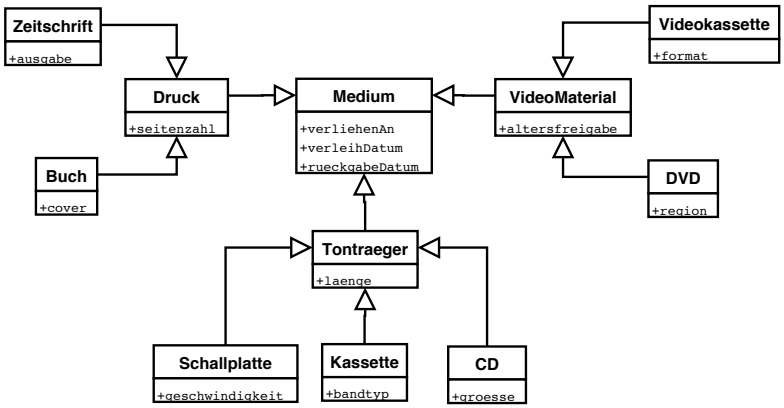

Abbildung 1: UML Diagram der Medien-Klassenhierarchie

Druck, Tonträger und Videomaeterial. Jede dieser Klassen hat wiederum Attribute, welche Sie sowohl von der Basisklasse als auch von den anderen abgeleiteten Klassen unterscheidet. So ist das Attribut Seitenzahl sicherlich nur für gedrucktes wirklich interessant und nicht etwa für Tonträger und Videomaterial. Dagegen besitzt die Klasse Videomaterial das Attribut altersfreigabe, was bei Büchern eher unüblich ist.
Von der Klasse Druck leiten sich nun wiederum die Klassen Buch und Zeitschrift ab, von der Klasse Tonträger die Klassen Schallplatte, Kassette und CD und von der Klasse Videomaterial die Klassen Videokasette und DVD. Diese erweitern ihre direkten Basisklassen auch um Attribute, die mehr oder weniger spezifisch für die abgeleitete Klasse sind. D.h. in dieser Klassenhierarchie geht man von einer sehr generellen Klasse aus und leitet davon dann spezielisierte Klassen ab, von denen wiederum Klassen abgeleitet werden, die noch spezialisierter sind.

Aufgabe 5:

Die sogenannte Standard Template Library in C++ stellt dem Programmierer eine Reihe von häufig verwendeten Strukturen (Klassen) zur Verfügung, darunter auch eine Klasse für Strings. Da diese sicherer ist und mehr Funktionalität bietet als die C String (**char** Felder) die wir bisher verwendet haben, wollen wir ab sofort diese Klasse anstelle der C Strings verwenden. Um C++ Strings zu verwenden muss man die Deklaration der Klasse in das

Programm einbinden, diese befinden sich in der Header Datei *string*, d.h. um C++ Strings in einem Programm zun verwenden muß man die Deklaration mit `#include <string>` in den Quellcode einbinden. Ein weiterer Vorteil von C++ Strings ist, daß man sich nicht im Vorraus Gedanken über dessen Länge machen muß. Ansonsten ist die Benutzung von C++ Strings der von C Strings sehr ähnlich. Man sollte jedoch beachten, daß C++ Strings im Gegensatz zu C Strings keine 0 am Ende haben. C++ Strings verfügen über viele Methoden, um Strings zu manipulieren und zu untersuchen.
Hier ein kleines Beispiel:

```
1  #include <string>
2  #include <iostream>
3
4  int main()
5  {
6      std::string name;
7      std::cout << "Bitte geben Sie Ihren Namen ein: ";
8      getline(std::cin,name);
9      std::cout << "Sie heissen: " << name << "\n";
10     std::cout << "Ihr Name hat: " << name.size() << " Buchstaben.\n";
11     std::cout << "erster Buchstabe: " << name[0]  << "\n";
12     std::cout << "letzter Buchstabe: " << name[name.size()-1]  << "\n";
13     return 0;
14 }
```

Zeile 1: Bindet die Deklaration für C++ Strings ein

Zeile 6: Deklariert eine Variable vom Type *std::string* mit dem Namen *name*. Die Variable ist somit eine Instanz der Klasse *std::string*. Instanzen werden durch den Standard Konstruktor initialisiert, d.h. nach Zeile 6 enthält der String *name* einen Leerstring.

Zeile 7: Fordert den Anwender dazu auf, seinen Namen einzugeben.

Zeile 8: Liest den Namen vom Anwender mit Hilfe der *getline* Funktion ein. Die Syntax ist aber diesmal eine etwas Andere. Zum ersten ist dieses *getline* keine Methode von *std::cin*, sondern eine Funktion, welche zwei Argumente übernimmt. Das erste Argument ist der sogenannte stream aus dem gelesen wird, in diesem Fall *std::cin*, und das zweite Argument ist das Objekt, in welchem das Ergebnis gespeichert wird. Da der C++ String *name* eine variable Länge hat, braucht man auch die Länge der einzulesenden Zeichen nicht angeben.

Zeile 9: Gibt den Namen aus.

Zeile 10 Gibt die Länge des Strings aus, welche man man mit Hilfe der Methode *size* der Klasse *string* ermittlen kann.

Zeile 11 und 12: Geben jeweils den ersten und letzten Buchstaben des Namens aus. Die einzelnen Buchstaben eines C++ Strings sind wiederum Daten vom Typ **char**, und man kann über den Namen der Stringvariablen mit nachgestellten Index in eckigen Klammern auf die einzelnen Zeichen zugreifen, genau wie bei C Strings. Der Index beginnt wiederum bei 0, d.h. der Index des letzten Zeichens ist `name.size()-1`.

Versuchen Sie die Aufgabe 1 und 2 vom zweiten Übungsblatt so umzuschreiben, daß diese nun C++ Strings verwenden.

```
1  bool istEineZahl(std::string s)
2  {
3    bool result=true;
4    if(s.size()>0)
5    {
6      if(s[0]<48)
7      {
8        result=false;
9      }
10     else if(s[0] > 57)
11     {
12       result=false;
13     }
14     else if(s.size()>1)
15     {
16       if(s[1] < 48)
17       {
18         result=false;
19       }
20       else if(s[1] > 57)
21       {
22         result=false;
23       }
24     }
25   }
26   else
27   {
28       result=false;
29   }
```

```
30   return result;
31 }
32
33 unsigned int umwandelnInZahl(std::string s)
34 {
35    unsigned int result=0;
36    if(s.size()>0)
37    {
38      result=s[0]-48;
39      if(s.size()>1)
40      {
41        result*=10;
42        result+=(s[1] - 48);
43      }
44    }
45    return result;
46 }
47
48 int main()
49 {
50   std::cout << "Bitte geben Sie eine zweistellige Zahl ein:";
51   std::string s;
52   std::getline(std::cin,s);
53   if(istEineZahl(s)==true)
54   {
55     unsigned int zahlenWert=umwandelnInZahl(s);
56     std::cout << "Sie haben die Zahl " << zahlenWert << " eingegeben.\n";
57   }
58   else
59   {
60     std::cout << "Es handelt sich bei der Eingabe nicht um eine Zahl.\n";
61   }
62   return 0;
63 }
```

Worauf man bei der Implementierung mit C++ Strings achten sollte ist, abzufragen, ob der String überhaupt genug Zeichen enthält bevor man versucht diese auszulesen. Dies macht man mit Hilfe der *size* Methode, welche die Anzahl der Zeichen im String zurückliefert.
Die Implementierung von Aufgabe 2 werde ich hier nicht anführen, da sie genau dieselben Funktionen enthalten sollte wie Aufgabe 1. Die zusätzliche Funktion zur Berechnung der Fakultät verwendet keine Zeichenketten.
Um Aufgabe 4 sinnvoll zu implementieren, sodass C++ Strings verwendet werden kann man z.B. die Funktion *binaerZahlAusgeben* so umschreiben,

dass sie die zahl nicht direkt ausgibt, sondern als String zurückliefert. Die Funktion könnte dann z.B. folgendermassen aussehen:

```
1  std::string binaerzahlAusgeben(unsigned int n)
2  {
3      std::string result;
4      unsigned int sizeInBits=(sizeof n) * 8;
5      unsigned int i;
6      for(i=sizeInBits; i>=1; --i)
7      {
8          unsigned int bitWert=wertVonBit(i);
9          if(n>=bitWert)
10         {
11             result+= "1";
12             n-=bitWert;
13         }
14         else
15         {
16             result+= "0";
17         }
18     }
19     return result;
20 }
```

Was man hier sieht ist, dass man an einen C++ String wie bei einer Addition Zeichen anhängen kann.