# Lösungen zu Übungsblatt 6

Ralph Gauges Ursula Rost Katja Wegner

28.11.2007

## Aufgabe 1:

Erweitern Sie die Klasse *Image* aus Aufgabe 1 vom 21.11.2007 so, dass der angeforderte Speicher in jedem Fall wieder freigegeben wird.
Erweitern Sie die Klasse um einen Kopierkonstruktor und einen Zuweisungsoperator.

```
1  Image::~Image()
2  {
3      delete[] data;
4  }
5
6  Image::Image(const Image& src)
7  {
8      this->width=src.width;
9      this->height=src.height;
10     unsigned int numChars=width * height * 3;
11     this->data=new unsigned char[numChars];
12     unsigned int i;
13     for(i=i;i<numChars;++i)
14     {
15         this->data[i]=src.data[i];
16     }
17 }
18
19 Image& Image::operator=(const Image& src)
20 {
21     delete[] this->data;
22     this->width=src.width;
23     this->height=src.height;
```

```
24      unsigned int numChars=width * height * 3;
25      this->data=new unsigned char[numChars];
26      unsigned int i;
27      for(i=i;i<numChars;++i)
28      {
29          this->data[i]=src.data[i];
30      }
31      return *this;
32  }
```

Wie in der Aufgabenstellung verlangt muss man 3 neue Methoden deklarieren. Auf die Deklarationen selbst wollen wir hier jedoch nicht weiter eingehen. Es sind hier lediglich die Implementierungen der neuen Methoden aufgeführt.

**Zeile 1-4:** Der Destruktor muss dafür sorgen, dass der dynamisch angeforderte Speicher wieder freigegeben wird. Hierbei muss man darauf achten, dass der Speicher mit **delete**[] freigegeben wird, da er mit **new**[] angefordert wurde.

**Zeile 6-17:** Der Kopierkonstruktor kopiert zuerst die Attribute für Höhe und Breite und berechnet dann, wieviel Speicher neu angefordert werden muss. Anschließend wird die berechnete Menge an Speicher für *data* angefordert. Nun müssen noch die eigentlichen Daten aus dem einen Bild in das andere kopiert werden. Dies erreicht man am Besten über eine einfache Schleife in der jedes Element einzeln kopiert wird.

**Zeile 19-30:** Der Zuweisungsoperator ist dem Kopierkonstruktor sehr ähnlich. Es gibt lediglich zwei Unterschiede. Erstens wird der Zuweisungsoperator auf eine Instanz von *Image* angewendet, die bereits existiert, d.h. für die bereits Speicher reserviert wurde. Dieser Speicher muss zuerst gelöscht werden bevor man neuen Speicher anfordert. Das Kopieren der Daten selbst funktioniert genau gleich wie im Kopierkonstruktor. Der zweite Unterschied ist der Rückgabewert der Methode. Der Kopierkonstruktor hat, wie alle Konstruktoren, keinen Rückgabewert; der Zuweisungsoperator gibt jedoch eine Referenz auf sich selbst zurück. Dies erreicht man, indem man *this* dereferenziert und dies zurückgibt.

# Aufgabe 2:

Erweitern Sie die Klasse *Liste* aus Aufgabe 2 vom 21.11.2007 so, dass der gesamte angeforderter Speicher in jedem Fall wieder freigegeben wird.

Schreiben Sie zusätzlich einen Kopierkonstruktor und einen Zuweisungsoperator für die Klasse.

```
1  Liste::~Liste()
2  {
3      delete[] elements;
4  }
5
6  Liste::Liste(const Liste& src)
7  {
8      this->listLength=src.listLength;
9      this->currentPosition=src.currentPosition;
10     unsigned int i;
11     for(i=0;i<this->listLength;++i)
12     {
13         this->elements[i]=src.elements[i];
14     }
15 }
16
17 Liste& Liste::operator=(const Liste& src)
18 {
19     delete[] this->elements;
20     this->listLength=src.listLength;
21     this->currentPosition=src.currentPosition;
22     unsigned int i;
23     for(i=0;i<this->listLength;++i)
24     {
25         this->elements[i]=src.elements[i];
26     }
27     return *this;
28 }
```

Die Lösung für diese Aufgabe entspricht im Grunde genau der Lösung aus Aufgabe 1 und bedarf deshalb keiner weiteren Erläuterung.

# 1 Aufgabe 3:

Schreiben Sie eine Klasse für Komplexe Zahlen.
Komplexe Zahlen treten z.B. auf wenn man die Quadratwurzel aus einer negativen Zahl zieht. Ähnlich wie die Klasse Bruch verfügen Komplexe Zahlen über zwei Komponenten, die aber diesmal beide als double Werte repräsentiert werden sollen. Der eine Wert ist der sogenannte Realteil der

Komplexen Zahl, der andere der sogenannte Imaginärteil. Die Darstellung einer solchen Komplexen Zahl sieht folgendermassen aus: $a + bi$.
Dabei ist $a$ der Realteil der Zahl und $b$ der Imaginärteil. Das Symbol $i$ steht für das Ergebnis der Quadratwurzel aus $-1$ ($\sqrt[2]{-1}$).
Detailiertere Informationen zu Komplexen Zahlen findet man in Mathematikbüchern, oder z.B. hier auf Wikipedia.
Die Klasse soll normale mathematische Operationen zwischen zwei Komplexen Zahlen ermöglichen. Überlegen Sie sich welche Konstruktoren sinnvoll wären und implementieren Sie diese ebenfalls.
Die Rechenregeln für Komplexe Zahlen sind wie folgt:

**Addition:** $(a + bi) + (c + di) = (a + c) + (b + d)i$

**Subtraction:** $(a + bi) - (c + di) = (a - c) + (b - d)i$

**Multiplikation:** $(a + bi) * (c + di) = (a * c) + (c * b)i + (a * d)i + (b * d)i^2 = (a * c - b * d) + (c * b + a * d)i$

**Division:** $\frac{a+bi}{c+di} = \frac{(a+bi)*(c+di)}{(c+di)*(c+di)} = \frac{ac+bd}{c^2+d^2} + \frac{bc-ad}{c^2+d^2}i$

Die Symbole a und c stehen hierbei immer für die Realteile der beiden Komplexen Zahlen und b und d für die Imaginärteile. $i$ repräsentiert $\sqrt[2]{-1}$.
Diese Klasse benötigen wir vermutlich in späteren Aufgaben, um die Mandelbrotmenge (Apfelmännchen) zu berechnen.

------- ComplexNumber.h -----

```
1  #ifndef COMPLEXNUMBER_H_
2  #define COMPLEXNUMBER_H_
3
4  class ComplexNumber
5  {
6      protected:
7          double realPart;
8          double imagPart;
9
10     public:
11         ComplexNumber();
12         ComplexNumber(double r,double i);
13         ComplexNumber& operator+=(const ComplexNumber& right);
14         ComplexNumber& operator-=(const ComplexNumber& right);
15         ComplexNumber& operator*=(const ComplexNumber& right);
```

```
16          ComplexNumber& operator/=(const ComplexNumber& right);
17          const ComplexNumber operator+(const ComplexNumber& right) const;
18          const ComplexNumber operator-(const ComplexNumber& right) const;
19          const ComplexNumber operator*(const ComplexNumber& right) const;
20          const ComplexNumber operator/(const ComplexNumber& right) const;
21 };
22
23 #endif // COMPLEXNUMBER_H_
```

**Zeile 7-8:** Deklariert zwei Attribute vom Typ **double**, um den Real- und den Imaginärteil der Komplexenzahl zu speichern.

**Zeile 11-12** Deklariert zwei Konstruktoren. Einen Standardkonstruktor, der keine Argumente entgegennimmt und einen Konstruktor, der zwei **double** Werte entgegennimmt. Ein Kopierkonstruktor, Destruktor und Zuweisungsoperator sind hier nicht nötig, da diese Klasse keinen Speicher dynamisch anfordert.

**Zeile 13-16:** Deklariert die vier mathematischen Zuweisungsoperatoren für `+=`,`-=`,`/=` und `*=`

**Zeile 17-20:** Deklariert die vier mathematischen Operatoren für `+`,`-`,`/` und `*`

------- ComplexNumber.cpp -----

```
1 ComplexNumber::ComplexNumber():realPart(0.0),imagPart(0.0){}
2
3 ComplexNumber::ComplexNumber(double r,double i):realPart(r),imagPart(i){}
4
5 ComplexNumber& ComplexNumber::operator+=(const ComplexNumber& right)
6 {
7     this->realPart += right.realPart;
8     this->imagPart += right.imagPart;
9     return (*this);
10 }
11
12 ComplexNumber& ComplexNumber::operator-=(const ComplexNumber& right)
13 {
14     this->realPart -= right.realPart;
15     this->imagPart -= right.imagPart;
16     return (*this);
17 }
18
```

```
19 ComplexNumber& ComplexNumber::operator*=(const ComplexNumber& right)
20 {
21     double temp1 = this->realPart*right.realPart - this->imagePart*right.imagPart;
22     double temp2 = this->realPart*right.imagPart + right.realPart*this.imagPart;
23     this->realPart = temp1;
24     this->imagePart = temp2;
25     return (*this);
26 }
27
28 ComplexNumber& ComplexNumber::operator/=(const ComplexNumber& right)
29 {
30     double divisor=right.realPart*right.realPart + right.imagPart*right.imagPart;
31     double temp1=this->realPart*right.realPart + this->imagePart*right.imagPart;
32     double temp2=this->imagePart*right.realPart-this->realPart*right.imagePart;
33     this->realPart=temp1/divisor;
34     this->imagPart=temp2/divisor;
35     return (*this);
36 }
37
38 const ComplexNumber ComplexNumber::operator+(const ComplexNumber& right) const
39 {
40     ComplexNumber result=(*this);
41     result+=right;
42     return result;
43 }
44
45 const ComplexNumber ComplexNumber::operator-(const ComplexNumber& right) const
46 {
47     ComplexNumber result=(*this);
48     result-=right;
49     return result;
50 }
51
52 const ComplexNumber ComplexNumber::operator*(const ComplexNumber& right) const
53 {
54     ComplexNumber result=(*this);
55     result*=right;
56     return result;
57 }
58
59 const ComplexNumber ComplexNumber::operator/(const ComplexNumber& right) const
60 {
61     ComplexNumber result=(*this);
62     result/=right;
63     return result;
```

```
64 }
```

**Zeile 1:** Der Standardkonstruktor initialisiert den Real- und den Imaginärteil der Komplexen Zahl mit 0.0.

**Zeile 3:** Der Konstruktor mit den zwei **double** Werten initialisiert den Real- und Imaginärteil der Komplexen Zahl mit den beiden angegebenen Werten.

**Zeile 5-36:** Implementiert die mathematischen Zuweisungsoperatoren gemäß der Rechenregeln, die in der Aufgabenstellung angegeben waren. Es ist zu beachten, dass jede der Methoden eine Referenz auf die eigene Instanz zurückgeben muss.

**Zeile 38-64:** Implementiert die mathematischen Operatoren gemäß den Rechenregeln aus der Aufgabenstellung. Jede dieser Methoden gibt ein neues Objekt zurück.