Lösungen zu Übungsblatt 7

Ralph Gauges Ursula Rost Katja Wegner

05.12.2007

Aufgabe 1:

Schreiben Sie einen Ausgabeoperator für die Klasse für Komplexe Zahlen aus Aufgabe 3 vom 28.11.2008.

Die Deklaration des Ausgabeoperators kann man in die Header Datei für die Klasse selbst packen. Am besten schreibt man sie hinter die Deklaration der Klasse. Die Definition packt man an eine beliebige Stelle in der Source Datei.

```
1 std::ostream& operator<<(std::ostream& os, const ComplexNumber& number)
2 {
3     os << number.getRealpart() << "+" << number.getImaginaryPart() << "i";
4     return os;
5 }
```

Zeile 1: Der Name der Methode, die man überschreiben muss, um einen Ausgabeoperator zu schreiben lautet **operator**<<. Die Methode nimmt zwei Argumente entgegen. Das erste Argument ist eine Referenz auf den Ausgabestrom, auf den die Ausgabe erfolgen soll, das zweite Argument ist eine konstante Referenz auf das auszugebende Objekt. Die Funktion gibt die Referenz auf den Ausgabestrom selbst wieder zurück. Dies ist notwendig, um mehrere Ausgaben mit << in einer Zeile schreiben zu können. Die Funktion ist als globale Funktion und nicht als Methode der Klasse *ComplexNumber* zu definieren. Dies sieht man schon allein daran, dass dem Methodennamen der Klassenname nicht als Präfix vorangestellt ist.

Zeile 3: Gibt den Realteil und den Imaginärteil der Zahl gefolgt vom Symbol "i" aus. Die beiden Teile werden durch ein "+" Zeichen getrennt. Die

Ausgabe selbst erfolgt wie gewohnt über den Ausgabeoperator `ii` auf den Ausgabestrom, den die Funktion als erstes Argument übergeben bekommt. Man sollte dabei beachten, dass die Methode für den Ausgabeoperator nur auf als **public** deklarierte Methoden und Attribute der auszugebenden Klasse zugreifen kann, d.h. man braucht in der Klasse *ComplexNumber* Methoden um den Realteil und den Imaginärteil einer Zahl auszulesen, da die Attribute selbst als **protected** oder **private** deklariert sind. Des Weiteren müssen diese Methoden als **const** Methoden deklariert sein, da die Methode für den Ausgabeoperator eine konstante Referenz auf das *ComplexNumber* Objekt übergeben bekommt.

Zeile 4: Gibt die Referenz auf den Ausgabestrom zurück.

Aufgabe 2:

Schreiben Sie einen Ausgabeoperator für die Klasse *Bruch* aus Aufgabe 2 vom 07.11.2007.

Auch hier schreibt man die Deklaration des Ausgabeoperators am besten in die Header Datei für die Klasse selbst und die Definition in die Source Datei der Klasse.

```
1 std::ostream& operator<<(std::ostream&,const Bruch& bruch)
2 {
3     os << bruch.getZaehler() << "/" << bruch.getNenner();
4     return os;
5 }
```

Die Implementierung ist genau entsprechend der Implementierung aus Aufgabe 1 und es bedarf deshalb keiner weiteren Erläuterung.

Aufgabe 3:

Erweitern Sie die Klasse *Image* aus Aufgabe 1 vom 28.11.2007 um eine Methode, welche das Bild als TGA Datei speichert.

TGA ist eines der einfachsten Formate, um Bitmap Grafiken zu speichern. Das TGA Format gibt es in mehreren unterschiedlichen Varianten. Wir verwenden hier die einfachste Variante, die lediglich aus einem sogenannten Hea-

Byte Nr.	Wert
0-1	0
2	2
3-11	0
12-13	Breite in Pixeln
14-15	Höhe in Pixeln
16	24
17	32

Tabelle 1: TGA Header

der besteht, welcher Informationen wie die Höhe und Breite des Bildes und den eigentlichen Bilddaten in unkomprimierter Form enthält.

Der Header kommt als erstes und hat, die in der Tabelle 1 angegebene, Struktur.

Was die einzelnen Zahlen bedeuten, soll uns hier nicht weiter interessieren. Die einzelnen Zahlen sind als Byte-Werte (auf den meisten Plattformen **char** Datentyp) aufzufassen, nur die Höhe und die Breite werden als 2 Byte Werte (Word) abgespeichert. Da das TGA Format vor allem für Rechner mit Intel Prozessoren entwickelt wurde, muss man eine kleine Besonderheit beim Abspeichern der Höhe und der Breite berücksichtigen. Word-Werte werden auf dieser Plattform in zwei Byte Werte aufgeteilt und diese werden in umgekehrter Reihenfolge geschrieben.

Angenommen das Bild hätte eine Breite von 1025 Pixeln, dies entspräche der Hexadezimalzahl 0x0401. Teilt man diese in zwei Byte Werte, so erhält man 0x04 und 0x01. Will man nun diese Zahl in die Datei schreiben, so muss man zuerst die 0x01 schreiben und dann die 0x04. Dies ist mit den Operatoren die heute behandelt wurden leicht möglich.

Im Anschluss an diesem Header folgen die Bilddaten und zwar jeweils als Rot-, Grün- und Blauwert, wobei jeder Wert ein Byte groß sein muss. D.h. Die Farbe Schwarz hätte die Werte 0 0 0 und die Farbe weiß hätte die Werte 255 255 255. Da dies auch das Format ist, das wir in der Image Klasse verwendet haben, können wir die Daten aus der Image Klasse einfach Byte für Byte in die Datei schreiben. Die Bilddaten werden dabei Zeilenweise von Oben nach Unten geschrieben.

```

1 void Image::writeHeader(std::ostream& os) const
2 {

```

```

3     char header[18]={0,0,2,0,0,0,
4                   0,0,0,0,0,0,
5                   (char)this->width, (char)(this->width >> 8),
6                   (char)this->height, (char)(this->height >> 8),
7                   24,32
8                   };
9
10    os.write(header,18);
11 }
12
13 void Image::writeData(std::ostream& os) const
14 {
15     os.write((char*)this->data, this->width * this->height * 3);
16 }
17
18 bool Image::saveTGA(const char* filename) const
19 {
20     bool success=false;
21     std::ofstream ofs(filename);
22     if(ofs.good())
23     {
24         writeHeader(ofs);
25         writeData(ofs);
26         ofs.close();
27         success=true;
28     }
29     return success;
30 }

```

Die Implementierung zum Schreiben der Bilddaten wurde der Übersichtlichkeit halber in drei Methoden aufgeteilt. Eine Methode schreibt den Header des Bildes, die zweite Methode schreibt die eigentlichen Bilddaten und die dritte Methode ruft die ersten beiden auf, um das Bild in eine Datei zu schreiben.

Zeile 1-11: Diese Methode bekommt eine Referenz auf einen Ausgabestrom übergeben. In der Methode selbst wird ein **char** Feld der Länge 18 deklariert und gleich mit den Werten aus der Tabelle gefüllt. Wie gesagt, muss man bei den Werten für die Breite und Höhe des Bildes die Daten so aufteilen, dass die niederwertigen 8 Bit zuerst geschrieben werden und dann erst die höherwertigen 8 Bit. Die niederwertigen 8 Bit erhält man einfach dadurch dass man den Wert auf einen **char** Datentyp castet, dies bewirkt, dass alle Bit über dem achten abgeschnitten werden. Andererseits bekommt man die höherwertigen 8 Bit indem man

alle Bits des Wertes zuerst acht mal nach rechts verschiebt, d.h. man schiebt die Bits 9-16 an die Stelle der Bits 1-8. Anschliessend castet man den Wert wiederum auf den **char** Datentyp um nur die Bits 1-8 auszugeben. (Eigentlich beginnt man beim Zählen der Bits mit der 0, d.h. korrekterweise müssten wir hier jeweils von den Bits 0-7 etc reden.) Das Verschieben und das Casten für die höherwertigen 8 Bit müssen in dieser Reihenfolge ausgeführt werden. Erst Casten und dann Schieben liefert ein falsches Ergebnis. Warum das so ist kann sich ja jeder selbst mal überlegen.

Diese Methode geht davon aus, dass der **char** Datentyp eine Breite von 8 Bit besitzt, was für vermutlich 99,9% aller Systeme OK sein sollte. Ist der Header gefüllt, kann man ihn mit der `write` Methode der Ausgabestrom Klasse schreiben. Die Methode bekommt einen Zeiger auf einen Speicherbereich aus **char** Daten übergeben, was der Header ist, und die Anzahl der zu schreibenden **char** Werte, in diesem Fall 18.

Zeile 13-16: Die Methode ist recht kurz, sie ruft einfach nur die `write` Methode des übergebenen Ausgabestroms auf, um die Bilddaten auszugeben. Da die Bilddaten bereits als Feld von **unsigned char** Werten abgespeichert sind, müssen wir auch nicht viel beachten. Da die Daten jedoch als Feld oder Zeiger auf **unsigned char** Daten gespeichert sind, möchte die `write` Methode jedoch eine Zeiger auf **char** haben, also müssen wir den Zeiger auf die Daten entsprechend casten. Da **unsigned char** und **char** Daten dieselbe Anzahl an Bits besitzen ist dies ohne weiteres möglich.

Zeile 18-30: Die Methode bekommt einen konstanten C-String übergeben, der den Namen der Datei enthält, in die die Daten geschrieben werden sollen. In der Methode wird versucht eine Instanz der Klasse `std::ofstream` mit dem Namen der Datei zu initialisieren, was einem Öffnen der Datei zum Schreiben gleichkommt. Ohne weitere Argumente würde eine existierende Datei mit demselben Namen einfach überschrieben und die alten Daten wäre weg, d.h. hier sollte man etwas aufpassen und gegebenenfalls zuerst überprüfen, ob es eine Datei mit demselben Namen bereits gibt. Nachdem versucht wurde, die Datei zu öffnen, muss man mit der Methode `good` überprüfen, ob dies auch funktioniert hat. Liefert die Methode den Wert **true** zurück, so kann man damit beginnen in die Datei zu schreiben.

Das Schreiben der Daten erfolgt mit Hilfe der oben beschriebenen Methoden. Korrekterweise sollte man eigentlich nach jedem Schreibvorgang überprüfen, ob der Status des Ausgabestromes weiterhin O.K. ist, anderenfalls sollte man eine entsprechende Fehlermeldung ausgeben. Nach dem Schreiben der Daten wird die Datei mit der *close* Methode geschlossen.

Hier auch nochmal die Methoden zum Setzen und Auslesen eines bestimmten Pixels. Die ursprüngliche Lösung hatte einen kleinen Fehler und außerdem wurden die Daten als RGB Werte, d.h. Rot-, Grün- und Blauwert in dieser Reihenfolge gespeichert. Da TGA Dateien die Daten jedoch in genau umgekehrter Reihenfolge speichern, also BGR, ist es einfacher, dies auch mit den Daten so zu machen. Nur so kann man die gesamten Bilddaten mit einem Funktionsaufruf (siehe oben) schreiben.

```

1 bool Image::setPixel(unsigned int x,unsigned int y,unsigned char red ,
2                       unsigned char green ,unsigned char blue)
3 {
4     if((x<width) && (y<height))
5     {
6         unsigned char* pos=data + y*width*3 +x*3;
7         *(pos)=blue;
8         *(++pos)=green;
9         *(++pos)=red;
10        return true;
11    }
12    else
13    {
14        return false;
15    }
16 }
17
18 bool Image::getPixel(unsigned int x,unsigned int y,unsigned char& red ,
19                    unsigned char& green ,unsigned char& blue)
20 {
21     if((x<width) && (y<height))
22     {
23         unsigned char* pos=data + y*width*3 +x*3;
24         blue=*pos;
25         green=*(++pos);
26         red=*(++pos);
27         return true;
28     }
29    else

```

```
30     {  
31     return false;  
32     }  
33 }
```