Lösungen zu Übungsblatt 9

Ralph Gauges Ursula Rost Katja Wegner

09.01.2008

Aufgabe 1:

Implementieren Sie die Klasse *Liste* aus Aufgabe 2 vom 28.11.2007 als Template Klasse, welche beliebige Datentypen speichern kann.
Die Klasse soll eine Ausnahme (Exception) erzeugen, wenn auf nicht existierende Elemente zugegriffen wird. Schreiben Sie ein kleines Testprogramm, dass die Liste einmal mit 10 beliebigen **double** Werten füllt und einmal mit C++ Strings und anschließend die Listen ausgibt.

——— Liste.h ———

```
1 #ifndef LISTE_H_
2 #define LISTE_H_
3
4 #include "exceptions.h"
5 #include <stdexcept>
6 #include <iostream>
7
8 template<typename T>
9 class Liste
10 {
11   private:
12      T *elements;
13      unsigned int listLength;
14      unsigned int currentPosition;
15
16   public:
17     Liste (){this->init(10);}
18     Liste (unsigned int numElements)
19     {
20       this->init(numElements);
```

```
21      }
22
23      ~Liste()
24      {
25        delete[] this->elements;
26      }
27
28      Liste(const Liste<T>& src)
29      {
30        this->listLength=src.listLength;
31        try
32        {
33          this->elements=new T[this->listLength];
34        }
35        catch(std::bad_alloc e)
36        {
37          std::cerr << "ERROR. Not enought memory for ";
38          std::cerr << this->listLength << " elements." << std::endl;
39          throw;
40        }
41        unsigned int i;
42        for(i=0;i<this->listLength;++i)
43        {
44          this->elements[i]=src.elements[i];
45        }
46        this->currentPosition=src.currentPosition;
47      }
48
49
50      Liste<T>& operator=(const Liste<T>& src)
51      {
52        delete[] this->elements;
53        this->listLength=src.listLength;
54        this->currentPosition=src.currentPosition;
55        unsigned int i;
56        for(i=0;i<this->listLength;++i)
57        {
58          this->elements[i]=src.elements[i];
59        }
60        return *this;
61      }
62
63
64      void init(unsigned int i) throw(std::invalid_argument,std::bad_alloc)
65      {
```

```
66        if(i==0)
67        {
68          throw std::invalid_argument("list can not have 0 arguments.");
69        }
70        try
71        {
72          this->elements = new T[i];
73        }
74        catch(std::bad_alloc e)
75        {
76          std::cerr << "ERROR. Not enough memory for ";
77          std::cerr << i << " elements." << std::endl;
78          throw;
79        }
80        this->listLength = i;
81        this->currentPosition = 0;
82      }
83
84      void insertElementAtEnd1(T element) throw (full_list)
85      {
86          if (this->currentPosition < this->listLength)
87          {
88            this->elements[this->currentPosition] = element;
89            this->currentPosition++;
90          }
91          else
92          {
93            throw full_list();
94          }
95      }
96
97      void insertElementAtEnd2(T element)
98      {
99        if (this->currentPosition > this->listLength)
100       {
101         this->extendList();
102       }
103       this->elements[this->currentPosition] = element;
104       this->currentPosition++;
105     }
106
107     void extendList() throw(std::bad_alloc)
108     {
109       int *newElements;
110       try
```

```
111         {
112           newElements = new int[this->listLength + 10];
113         }
114         catch(std::bad_alloc e)
115         {
116           std::cerr << "ERROR. Not enough memory for ";
117           std::cerr << this->listLength << " elements." << std::endl;
118           throw;
119         }
120         unsigned int i;
121         for (i=0;i<currentPosition;i++)
122         {
123           newElements[i] = this->elements[i];
124         }
125         listLength += 10;
126         for (i=this->currentPosition;i< this->listLength; i++)
127         {
128           newElements[i] = 0;
129         }
130         delete[] this->elements;
131         this->elements = newElements;
132       }
133
134
135       void insertSorted(T element)
136       {
137         if (this->currentPosition > this->listLength)
138         {
139           this->extendList();
140         }
141         unsigned int pos = 0;
142         for (; (pos <= this->currentPosition)
143                 && (element > this->elements[pos]); ++pos)
144         {
145         }
146         if (pos < this->currentPosition)
147         {
148           unsigned int k;
149           for (k = this->currentPosition;k > pos; k--)
150           {
151             this->elements[k] = this->elements[k-1];
152           }
153           this->elements[pos] = element;
154           this->currentPosition++;
155         }
```

```
156         else
157         {
158           this->insertElementAtEnd1(element);
159         }
160       }
161
162
163       void deleteElementFromEnd() throw(empty_list)
164       {
165         if (this->currentPosition >= 0)
166         {
167           this->elements[this->currentPosition] = 0;
168           this->currentPosition--;
169         }
170         else
171         {
172           throw empty_list();
173         }
174       }
175
176
177       const T operator[](unsigned int i) const throw(std::out_of_range)
178       {
179         if(i>currentPosition)
180         {
181           throw std::out_of_range("index out of range");
182         }
183         return this->elements[i];
184       }
185
186       inline unsigned int getLength() const{return this->listLength;}
187       inline unsigned int getPosition() const{return this->currentPosition;}
188 };
189
190 template<typename T>
191 std::ostream& operator<<(std::ostream& os,const Liste<T>& liste)
192 {
193    unsigned int i;
194    for (i = 0; i < liste.getPosition(); i++)
195    {
196      os << "element at position " << i << ": " << liste[i] << std::endl;
197    }
198    return os;
199 }
200
```

```
201 #endif // LISTE_H_
```

Da Liste nun als Template implementiert ist, befindet sich der gesammte Code in der Header Datei. Da die generelle Funktionalität der Klasse sich nicht geändert hat, gehen wir hier nur auf die Punkte ein, die für die Template Klasse von Interesse sind.

Zeile 8: Die Klasse Liste wird als Template Klasse deklariert. Die deklariert einen Template Parameter für den Datentyp, der für die zu speichernden Elemente, benutzt werden soll.

Zeile 12: Das Attribut *elements* ist nun nicht mehr ein Zeiger auf **int**, sondern ein Zeiger auf den Datentyp, welcher durch den Template Parameter festgelegt wird.

Zeile 28-47: Der Kopierkonstruktor sieht nun etwas anders aus. Erstens bekommt er als Argument eine Referenz auf die Template Klasse *Liste*<*T*>, außerdem wird bei der Anforderung von Speicher mit **new** Speicher für Objekte vom Typ *T* angefordert. Um den Umgang mit Exceptions nochmals zu veranschaulichen, wird der Speicher hier innerhalb eines **try** Blocks angefordert. Falls nicht genügend Speicher vorhanden ist, wird eine Exception (*bad_alloc* ausgelöst. Diese wird abgefangen, eine Fehlermeldung ausgegeben und anschließend die Exception neu ausgelöst.

Zeile 50: Methoden, welche früher Referenzen auf *Liste* zurückgegeben haben, geben nun Referenzen auf die Template Klasse zurück. Entsprechend nehmen Methoden, welche vorher *List* Objekte oder Referenzen auf *List* Objekte übergeben bekamen, nun Objekte auf die Template Klasse oder Referenzen auf diese entgegen.

Zeile 64-82: Die *init* Methode löst nun eine Exception aus, wenn versucht wird eine Liste der Länge 0 zu initialisieren. Außerdem gibt die Methode eine Fehlermeldung aus, wenn nicht genügend Speicher zur Verfügung steht.

Zeile 84-95: *insertElementAtEnd1* löst eine Exception aus, wenn die Liste bereits voll ist.

Zeile 107-122: *extendList* gibt nun eine Fehlermeldung aus, wenn nicht genug Speicher vorhanden ist um die Liste zu erweitern.

Zeile 163-174: *deleteElementFromEnd* löst eine Exception aus, wenn man versucht ein Element aus einer bereits leeren Liste zu löschen.

Zeile 177-184: Die *getElementAt* Funktion wurde durch den Indexoperator ersetzt. Der Operator löst nun eine Exception aus, wenn versucht wird auf ein Element zuzugreifen, welches es nicht gibt.

Zeile 190-199: Anstelle der *print* Methode besitzt die Implementierung nun einen Ausgabeoperator für die Template Klasse, d.h. auch dieser muss nur einmal geschrieben werden und funktioniert für alle möglichen Instanzen der Template Klasse. Der Operator benutzt den neuen Indexoperator, um auf die einzelnen Elemente zuzugreifen.

exceptions.h

```
1 #ifndef EXCEPTIONS_H__
2 #define EXCEPTIONS_H__
3
4 #include <stdexcept>
5
6 class empty_list : public std::runtime_error
7 {
8   public:
9     empty_list():std::runtime_error("the list is already empty."){};
10 };
11
12 class full_list : public std::runtime_error
13 {
14   public:
15     full_list():std::runtime_error("the list is already full."){};
16 };
17 #endif // EXCEPTIONS_H__
```

Die Datei *exceptions.h* implementiert zwei von *std::runtime_error* abgeleitete Klassen, welche von der *Liste* Klasse als Exceptions benutzt werden. Beide Klassen implementieren lediglich einen Konstruktor, welches den Konstruktor der Oberklasse aufruft und eine Fehlermeldung setzt.

main.cpp

```
1 #include "Liste.h"
2 #include <string>
3 #include <iostream>
4
```

```
5  int main()
6  {
7    Liste<double> lDouble;
8    lDouble.insertElementAtEnd1(0.0);
9    lDouble.insertElementAtEnd1(1.1);
10   lDouble.insertElementAtEnd1(2.2);
11   lDouble.insertElementAtEnd1(3.3);
12   lDouble.insertElementAtEnd1(4.4);
13   lDouble.insertElementAtEnd1(5.5);
14   lDouble.insertElementAtEnd1(6.6);
15   lDouble.insertElementAtEnd1(7.7);
16   lDouble.insertElementAtEnd1(8.8);
17   lDouble.insertElementAtEnd1(9.9);
18   std::cout << lDouble;
19   Liste<std::string> lString;
20   lString.insertElementAtEnd1(std::string("Apfel"));
21   lString.insertElementAtEnd1(std::string("Orange"));
22   lString.insertElementAtEnd1(std::string("Birne"));
23   lString.insertElementAtEnd1(std::string("Erdbeere"));
24   lString.insertElementAtEnd1(std::string("Ananas"));
25   lString.insertElementAtEnd1(std::string("Zitrone"));
26   lString.insertElementAtEnd1(std::string("Limette"));
27   lString.insertElementAtEnd1(std::string("Banane"));
28   lString.insertElementAtEnd1(std::string("Feige"));
29   lString.insertElementAtEnd1(std::string("Dattel"));
30   std::cout << lString;
31   return 0;
32 }
```

In der *main* Funktion wird zuerst die Template Klasse *Liste* für den Datentyp **double** instanziiert. Anschließend werden 10 **double** Werte an die Liste angehängt und dann die Liste über den Ausgabeoperator ausgegeben. Das Ganze wird dann noch einmal durchgeführt, diesmal jedoch für C++ Strings anstelle der **double** Werte.

Aufgabe 2:

Schreiben Sie die Klasse für die komplexen Zahlen aus Aufgabe 2 vom 09.01.2008 so um, dass der Gleitkommadatentyp, in dem der Real- und der Imaginärteil der komplexen Zahl gespeichert werden, als Template-Argument deklariert ist.
Schreiben Sie anschließend auch die Funktionen bzw. Klassen zur Berechnung

der Mandelbrotmenge so um, dass sie diese neue Klasse benutzen und dass alle anderen Gleitkommawerte die für die Berechnung notwendig sind auch als Template-Argumente angegeben werden können.
Berechnen Sie nun die Mandelbrotmenge je einmal für komplexe **float** Zahlen, für komplexe **double** Zahlen und für komplexe **long double** Zahlen, indem Sie den gewünschten Datentyp als Template Argument benutzen. Wenn alles richtig war, sollten die drei Bilder für die Mandelbrotmenge fast identisch sein.

——— ComplexNumber.h ———-

```
1  #ifndef COMPLEXNUMBER_H_
2  #define COMPLEXNUMBER_H_
3
4  #include <iostream>
5  #include "DivisionByZero.h"
6
7  template<typename T>
8  class ComplexNumber
9  {
10   protected:
11     T realPart;
12     T imagPart;
13
14   public:
15     ComplexNumber():realPart(0.0),imagPart(0.0){}
16
17     ComplexNumber(T r,T i):realPart(r),imagPart(i){}
18
19     ComplexNumber<T>& operator+=(const ComplexNumber<T>& right)
20     {
21       this->realPart += right.realPart;
22       this->imagPart += right.imagPart;
23       return (*this);
24     }
25
26     ComplexNumber<T>& operator-=(const ComplexNumber<T>& right)
27     {
28       this->realPart -= right.realPart;
29       this->imagPart -= right.imagPart;
30       return (*this);
31     }
32
33     ComplexNumber<T>& operator*=(const ComplexNumber<T>& right)
```

```
34      {
35        T temp1 = this->realPart*right.realPart;
36        temp1 -= this->imagPart*right.imagPart;
37        T temp2 = this->realPart*right.imagPart;
38        temp2 += right.realPart*this->imagPart;
39        this->realPart = temp1;
40        this->imagPart = temp2;
41        return (*this);
42      }
43
44      ComplexNumber<T>& operator/=(const ComplexNumber<T>& right)
45           throw(division_by_zero)
46      {
47        T divisor=right.realPart*right.realPart;
48        divisor += right.imagPart*right.imagPart;
49        T temp1=this->realPart*right.realPart;
50        temp1 += this->imagPart*right.imagPart;
51        T temp2=this->imagPart*right.realPart;
52        temp2 -= this->realPart*right.imagPart;
53        if(divisor==0.0)
54        {
55          throw division_by_zero();
56        }
57        this->realPart=temp1/divisor;
58        this->imagPart=temp2/divisor;
59        return (*this);
60      }
61
62      const ComplexNumber<T> operator+(const ComplexNumber<T>& right) const
63      {
64        ComplexNumber<T> result=(*this);
65        result+=right;
66        return result;
67      }
68
69      const ComplexNumber<T> operator-(const ComplexNumber<T>& right) const
70      {
71        ComplexNumber<T> result=(*this);
72        result-=right;
73        return result;
74      }
75
76      const ComplexNumber<T> operator*(const ComplexNumber<T>& right) const
77      {
78        ComplexNumber<T> result=(*this);
```

```
79        result*=right;
80        return result;
81      }
82
83      const ComplexNumber<T> operator/(const ComplexNumber<T>& right)a
84         const throw(division_by_zero)
85      {
86        ComplexNumber<T> result=(*this);
87        try
88        {
89          result/=right;
90        }
91        catch(division_by_zero)
92        {
93          throw;
94        }
95        return result;
96      }
97
98      T getRealPart() const
99      {
100       return this->realPart;
101     }
102
103     T getImaginaryPart() const
104     {
105       return this->imagPart;
106     }
107 };
108
109 template<typename T>
110 std::ostream& operator<<(std::ostream& os,const ComplexNumber<T>& number)
111 {
112   os << number.getRealPart() << "+" << number.getImaginaryPart() << "i";
113   return os;
114 }
115
116 #endif // COMPLEXNUMBER_H_
```

Zeile 7: Die Klasse *ComplexNumber* wird als Template Klasse mit einem Template Argument deklariert. Das Template Argument dient dazu festzulegen, welcher Datentyp zur Speicherung des Real- und des Imaginärteils der Zahl verwendet wird.

Zeile 11-12: Die Attribute für den Real- und den Imaginärteil der Zahl

haben nun den Typ des Template Parameters aus Zeile 7.

15-107: Alle Vorkommen des Typs **double** wurden durch den Template Parameter T ersetzt. Alle Funktionen, die bisher eine Instanz oder eine Referenz auf eine Instanz von ComplexNumber zurücklieferten, oder als Argument entgegen nahmen, benutztem nun eine Instanz bzw. eine Referenz auf die Template Klasse.

Zeile 109-114: Der Ausgabeoperator wurde so geändert, dass er mit der neuen Template Klasse funktioniert, d.h. der Ausgabeoperator ist nun selbst eine Template Funktion. Der Template Parameter der Funktion bestimmt auch hier den Datentyp, den *ComplexNumber* verwendet. Es ist jedoch keineswegs notwendig, den Template Parameter hier auch T zu nennen, er könnte auch jeden anderen gültigen Namen haben.

——— main.cpp ———-

```
1  #include "ComplexNumber.h"
2  #include "Image.h"
3
4  #include <math.h>
5  #include <limits>
6
7
8  void createImage(Image& image,unsigned long int* data)
9  {
10   unsigned int i,iMax=image.getWidth()*image.getHeight();
11   unsigned long int iterations,maxIterations=0;
12   unsigned long int minIterations=std::numeric_limits<unsigned long int>::max();
13   for(i=0;i<iMax;++i)
14   {
15     maxIterations=(maxIterations > data[i])?maxIterations:data[i];
16     minIterations=(minIterations < data[i])?minIterations:data[i];
17   }
18   iMax=image.getHeight();
19   unsigned int j,jMax=image.getWidth();
20   unsigned char value;
21   double range=log10((double)maxIterations/(double)minIterations);
22   for(i=0;i<iMax;++i)
23   {
24     for(j=0;j<jMax;++j)
25     {
26       iterations=data[i*jMax+j];
27       value=(iterations==0)?255:255-(unsigned char)((log10((double)iterations
```

```
28                                                  /(double)minIterations)/range)*255.0);
29          image.setPixel(j,i,value,value,value);
30        }
31      }
32  }
33
34  template<typename T,unsigned long int MAX_ITERATIONS>
35  int calculate_mandelbrot(const char* filename,unsigned int width,
36                           unsigned int height,T minX,T maxX,T minY,T maxY)
37  {
38    T maxBetrag=4.0;
39    T stepSizeX=(maxX-minX)/T(width);
40    T stepSizeY=(maxY-minY)/T(height);
41    ComplexNumber<T> c;
42    ComplexNumber<T> z;
43    Image image(width,height);
44    unsigned int i,j;
45    unsigned long int* data=new unsigned long int[width*height];
46    for(i=0;i<height;++i)
47    {
48      for(j=0;j<width;++j)
49      {
50       z=ComplexNumber<T>(0.0,0.0);
51       T betrag=0.0;
52       c=ComplexNumber<T>(minX+j*stepSizeX,minY+i*stepSizeY);
53       unsigned long int k;
54       for(k=0;k < MAX_ITERATIONS && betrag < maxBetrag;++k)
55       {
56         z=z*z+c;
57         betrag=z.getRealPart()*z.getRealPart();
58         betrag+=z.getImaginaryPart()*z.getImaginaryPart();
59       }
60       data[i*width+j]=k;
61     }
62    }
63    createImage(image,data);
64    image.saveTGA(filename);
65    return 0;
66  }
67
68  int main()
69  {
70    const unsigned long int MAX_ITERATIONS=100000;
71    unsigned int imageWidth=600;
72    unsigned int imageHeight=400;
```

```
73    long double minX=-2.0;
74    long double maxX=1.0;
75    long double minY=-1.0;
76    long double maxY=1.0;
77    calculate_mandelbrot<float,MAX_ITERATIONS>
78    ("Mandelbrot_full_float.tga",imageWidth,imageHeight,minX,maxX,minY,maxY);
79
80    calculate_mandelbrot<double,MAX_ITERATIONS>
81    ("Mandelbrot_full_double.tga",imageWidth,imageHeight,minX,maxX,minY,maxY);
82
83    calculate_mandelbrot<long double,MAX_ITERATIONS>
84    ("Mandelbrot_full_longdouble.tga",imageWidth,imageHeight,minX,maxX,minY,maxY);
85
86    minX=-0.74364388706280050000;
87    maxX=-0.74364388701150150000;
88    minY= -0.13182590423097950000;
89    maxY=-0.13182590417968050000;
90    calculate_mandelbrot<float,MAX_ITERATIONS>
91    ("Mandelbrot_part2_float.tga",imageWidth,imageHeight,minX,maxX,minY,maxY);
92
93    calculate_mandelbrot<double,MAX_ITERATIONS>
94    ("Mandelbrot_part2_double.tga",imageWidth,imageHeight,minX,maxX,minY,maxY);
95
96    calculate_mandelbrot<long double,MAX_ITERATIONS>
97    ("Mandelbrot_part2_longdouble.tga",imageWidth,imageHeight,minX,maxX,minY,maxY);
98    return 0;
99 }
```

main.cpp wurde gegenüber der alten Version etwas geändert. Diese Änderungen waren notwendig, um die Berechnung für unterschiedliche Gleitkommadatentypen über Templates zu ermöglichen. Da die *main* Funktion keine Template Funktion sein kann, muss die Berechnung der Mandelbrotmenge in eine separate Funktion ausgelagert werden, welche als Template Funktion implementiert werden kann. (Zumindest macht dies die Aufgabe einfacher.) Außerdem wurde die Art geändert, wie die Bildpunkte eingefärbt werden. Die berechneten Bilder sind jetzt nicht mehr nur zweifarbig, sonder werden als Graustufenbilder abgespeichert. Je mehr Iterationen benötigt werden bevor der Grenzwert überschritten wird, desto dunkler wird der Punkt eingefärbt.

Zeile 8-32: Die *createImage* Methode generiert die Grafik aus einem Datensatz, der die Anzahl der ausgeführten Iterationen für jeden Punkt enthält. Die Methode sucht zunächst die höchste und die niedrigste Zahl an Iterationen, die der Datensatz enthält. Dadurch lassen sich

im nächsten Schritt den einzelnen Bildpunkten besser Graustufenwerte zuweisen. Es stehen 256 Graustufenwerte zur Verfügung, diese werden möglichst gleichmäßig verteilt, basierend auf der Anzahl der Iterationen eines Punktes. Dies ist nur eine von vielen Methoden, wie man den Punkten eine Farbe zuweisen kann. Die so entstandenen Bilder zeigen feinere Strukturen, als die zweifarbigen Bilder, die wir bisher berechnet haben.

Zeile 34-66: Die Template Funktion *calculate_mandelbrot* berechnet die Mandelbrotmenge für einen gegebenen Auschnitt aus der Zahlenebene. Die Template Funktion besitzt zwei Template Parameter. Der erste Parameter ist ein Datentyp und er gibt an, mit welchem Datentyp die Template Klasse *ComplexNumber* instanziiert werden soll. Der zweite Template Parameter ist kein Datentyp, sondern ein einfacher Wert. Der Wert legt die maximale Anzahl der Iterationen für die Methode fest.

Die Funktion selbst ist gegenüber der ursprünglichen *main* Funktion kaum verändert. Es gibt zwei wesentliche Unterschiede. Erstens sind alle Vorkommen des Datentyps **double** durch den Template Parameter T ersetzt. Zweitens setzt die Funktion nicht mehr direkt die Farbe für jeden Punkt des Bildes, sonder die Funktion speichert die Anzahl der Iterationen für jedem Punkt in einem Feld und übergibt dieses Feld zusammen mit der Instanz der Klasse *Image* an die oben beschriebene Funktion *createImage*. Diese Funktion erzeugt aus den übergebenen Daten die eigentlichen Bilddaten.

Nach der Erzeugung der Bilddaten wird das Bild schließlich in einer Datei gespeichert. Den Namen der Datei übergibt man der Funktion *calculate_mandelbrot* als erstes Argument. Die Anzahl der maximalen Iterationen über einen Template Parameter festzulegen ist in diesem Fall eher unnötig und bietet keinerlei Vorteile. Man hätte die Zahl genauso gut der Funktion als Parameter übergeben können. Es sollte hier lediglich dazu dienen die Verwendung von Werten als Template Parametern zu veranschaulichen. Es ist auch zu beachten, dass in dieser Variante für jede Instanziierung der Funktion mit einer anderen Zahl für *MAX_ITERATION* eine neue Funktion vom Compiler erzeugt wird. Dies ist nicht der Fall, wenn man den Wert als Funktionsargument übergibt.

Zeile 68-99: Die *main* Funktion berechnet die Mandelbrotmenge für zwei

Bereiche. Der erste Bereich ist derselbe wie in der ursprünglichen Aufgabe und er umfasst den größten Teil der Mandelbrotmenge. Die Mandelbrotmenge wird dreimal berechnet, einmal mit komplexen Zahlen, die **float** Werte benutzen, einmal für **double** Werte und noch einmal für **long double** Werte. Die Unterschiede in den so berechneten Bildern sind recht gering. Dasselbe wird für einen zweiten Bereich gemacht. Der zweite Bereich ist ein kleiner Ausschnitt aus dem ersten Bereich. Der Ausschnitt liegt am Rand der Mandelbrotmenge und ist so klein, dass die Strukturen dort mit der Genauigkeit des **float** Datentyps nicht mehr zu berechnen sind.