Lösungen zum Übungsblatt 5

Ralph Gauges Ursula Rost Katja Wegner

21.11.2007

Aufgabe 1:

Schreiben Sie eine Klasse *Image*, welche die Daten eines Bitmap-Bildes speichern soll. Die Größe des Bildes soll über den Konstruktor festlegbar sein und anschließend nicht mehr verändert werden können. Jeder Pixel der Bitmap soll einen Wert für die rote, grüne und blaue Komponente verfügen, wobei der Wertebereich für die einzelnen Farbkomponenten zwischen 0-255 liegen sollen, d.h. **unsigned char** genügt. Die Klasse soll ferner über Methoden verfügen mit denen man die Farbwerte der einzelnen Pixel setzen und auslesen kann. Da solche Bilder unter Umständen recht groß werden können, soll der Speicher für die Daten dynamisch verwaltet werden.

Wir werden diese Klasse in späteren Aufgaben erweitern und wenn Sie fertig ist in einem größeren Programm benutzen.

----- Image.h -----

```
1 #ifndef IMAGE_H_
2 #define IMAGE_H_
3
4 class Image
5 {
6     protected:
7         unsigned int width;
8         unsigned int height;
9         unsigned char* data;
10
11     public:
12         Image();
13         Image(unsigned int _width, unsigned int _height);
14         bool setPixel(unsigned int x, unsigned int y,
15                       unsigned char red,
```

```

16             unsigned char green ,
17             unsigned char blue);
18
19     bool getPixel(unsigned int x,unsigned int y,
20                 unsigned char& red ,
21                 unsigned char& green ,
22                 unsigned char& blue);
23 };
24
25 #endif // IMAGE_H_

```

Zeile 6-9: Deklariert die Attribute für die Höhe und Breite des Bildes und ein Attribut *data* als Zeiger auf die Bilddaten.

Zeile 12-22: Deklariert einen Standardkonstruktor, einen Konstruktor, welcher die Höhe und Breite des Bildes entgegennimmt und Methoden zum Setzen und Auslesen einzelner Pixel des Bildes. Die Methode zum Auslesen der Farbwerte eines Pixels übergibt drei formale Parameter als Referenzen an die Methode, um den Rot-, Grün- und Blau-Wert eines Pixels mit einem Methodenaufruf zu erhalten.

Der Speicher für die Bilddaten muss in diesem Fall mit dem **new**[] Operator reserviert werden. Eine entsprechende Deklaration eines Feldes in der Form `char data[width][height]`; ist nicht erlaubt. Der Wert für die Größe der Dimension eines Feldes muss bereits beim Kompilieren des Programms feststehen. In unserem Fall ist es aber möglich die Bildgröße bei der Ausführung des Programms festzulegen (Konstruktor mit Argumenten), die Variablen *width* und *height* dürfen also nicht zur Festlegung der Größe eines Feldes verwendet werden.

```

----- Image.cpp -----
1 #include "Image.h"
2
3 Image::Image():width(100),height(100),
4     data(new unsigned char[100*100*3])
5 {
6 }
7
8 Image::Image(unsigned int _width,unsigned int _height)
9     :width(_width),height(_height),
10     data(new unsigned char[_width*_height*3])
11 {

```

```

12 }
13
14 bool Image::setPixel(unsigned int x,unsigned int y,
15                     unsigned char red ,
16                     unsigned char green ,
17                     unsigned char blue)
18 {
19     if((x<width) && (y<height))
20     {
21         unsigned char* pos=data + y*width*3 +x*3;
22         *pos=red;
23         *(pos++)=green;
24         *(pos++)=blue;
25         return true;
26     }
27     else
28     {
29         return false;
30     }
31 }
32
33 bool Image::getPixel(unsigned int x,unsigned int y,
34                     unsigned char& red ,
35                     unsigned char& green ,
36                     unsigned char& blue)
37 {
38     if((x<width) && (y<height))
39     {
40         unsigned char* pos=data + y*width*3 +x*3;
41         red=*pos;
42         green=*(pos++);
43         blue=*(pos++);
44         return true;
45     }
46     else
47     {
48         return false;
49     }
50 }

```

Zeile 3-5: Implementation des Standardkonstruktors. Der Standardkonstruktor setzt die Breite und Höhe des Bildes auf jeweils 100 Pixel und fordert anschließend mit `new[]` Speicher an, um die Bilddaten für ein Bild der Größe 100x100 zu speichern. Für die einzelnen Farbwerte wird hier der

Datentyp **unsigned char** benutzt dessen Wertebereich auf den meisten Systemen zwischen 0 und 255 liegt. Der Speicherbedarf für ein Bild berechnet sich aus dem Produkt aus der Breite, der Höhe und dem Speicherbedarf pro Pixel, der in diesem Fall 3x die Größe eines **char** Wertes hat.

Zeile 8-12: Konstruktor, der die Größe des Bildes als Argumente entgegennimmt und entsprechend Speicherplatz reserviert.

Zeile 14-31: Methode zum Setzen der Farbwerte eines bestimmten Pixels. Ein Bild lässt sich als zweidimensionales Feld mit Spalten und Zeilen interpretieren, d.h. die Anzahl der Spalten entspricht der Breite des Bildes und die Anzahl der Zeilen der Höhe. Entsprechend kann jeder Pixel des Bildes über seinen Spalten und Zeilenindex angesprochen werden. Die Methode überprüft, ob die angegebenen Indizes für die Zeile und Spalte im gültigen Bereich liegen, ist dies nicht der Fall würden wir die Werte in irgendeinen Speicherbereich schreiben. Liegen die Indizes nicht im gültigen Bereich liefert die Methode den Wert **false** zurück. Liegen die Werte im gültigen Bereich berechnet die Methode die erste Position des Pixels im *data* Feld. Der Pixel mit den Indizes *x* und *y* liegt in Zeile *y* und Spalte *x*, d.h. man muss zum Startindex der Bilddaten $y * width * 3 + x * 3$ addieren, um die Adresse des Pixels zu erhalten. Die Farbwerte werden üblicherweise in der Reihenfolge Rot, Grün und Blau gesetzt. D.h. wir setzen zuerst den roten Farbwert an der berechneten Adresse und addieren dann eins zur Adresse, um zur Adresse des grünen Farbwertes des Pixels zu gelangen und setzen diesen. Entsprechendes gilt für den Blauwert des Pixels.

Zeile 33-50: Die Methode dient zum Auslesen der Farbwerte eines bestimmten Pixels. Die Indizes des Pixels werden wiederum überprüft, um sicherzustellen, dass wirklich Bilddaten gelesen werden, anschließend berechnen wir wiederum die Adresse des Pixels im Speicher und setzen den Rot-, Grün- und Blauwert des Pixels auf die formalen Parameter die der Methode übergeben wurden.

In diesem Beispiel fordern wir Speicher über den **new[]** Operator an, wir geben den Speicher aber niemals wieder frei, d.h. der Speicher würde erst beim Beenden des Programms wieder freigegeben. Wie man dies korrigieren kann, müssen wir noch behandeln.

Aufgabe 2:

Ziel ist die Implementierung einer Listenklasse welche Integer Werte speichern kann. Das Einfügen und Löschen von Elementen soll stets am Ende erfolgen.

Um die Liste von Elementen mit Hilfe eines Feldes variabler Größe zu realisieren, sollen Sie dynamische Felder verwenden.

Deklariieren Sie das Feld als Pointer:

```
int *elements;
```

Dann können Sie mit Hilfe des `new[]` Operators zur Laufzeit dem Array eine frei wählbare Größe geben:

```
int k = 10;
elements = new int[k];
```

Schreiben Sie zunächst die Header Datei und vereinbaren Sie dort folgende Attribute und Methoden:

- ein privates **elements** Attribut, welches, wie oben beschrieben, ein Feld von int Werten über einen Pointer definiert
- ein privates **listLength** Attribut, welches zur Laufzeit jeweils die aktuelle Listenlänge enthalten soll
- ein privates **currentPosition** Attribut, welches zur Laufzeit jeweils die Position enthält, an der die nächste Einfügung stattfinden wird
- eine private **init** Methode, die einen **unsigned int** Parameter hat und nichts zurückgibt
- eine public **insertElementAtEnd1** Methode mit einem **int** Parameter
- eine public **deleteElementFromEnd** Methode ohne Parameter
- eine public **printElementAt** Methode mit einem **int** Parameter
- eine public **printList** Methode ohne Parameter
- einen Konstruktor ohne Parameter (Standardkonstruktor)

- einen Konstruktor, der einen `unsigned int` Parameter hat

Implementieren Sie nun die Methoden der Klasse in der zugehörigen Quell-datei mit der folgenden Funktionalität:

init soll dem Attribut **elements** ein Feld von `int` Werten zuweisen, wobei die Anzahl der Werte durch den Parameter der Methode bestimmt wird (siehe oben). Die Elemente sollen alle auf 0 gesetzt werden. Darüber hinaus müssen die Attribute **listLength** und **currentPosition** entsprechend initialisiert werden (Hinweis: angefangen wird bei Position 0)

insertElementAtEnd1 soll ein Element an der aktuellen Position einfügen und den Wert für die aktuelle Position anschließend entsprechend aktualisieren. Falls die Liste (d.h. das Array) bereits voll ist, soll eine passende Fehlermeldung ausgegeben werden.

deleteElementFromEnd soll das Element an der aktuellen Position löschen, sofern die Liste nicht leer ist

printElementAt falls eine gültige Position angegeben wurde (zwischen Anfang und Ende der Liste) soll dieses Element mit seiner Position zusammen formatiert ausgegeben werden (z.B. in der Form: 'element at position : 27')

printList soll die ganze Liste formatiert ausgegeben und dazu die vorige Methode benutzen

Standardkonstruktor soll die Liste mit 10 Elementen initialisieren

Konstruktor mit Parameter soll die Liste mit der Anzahl von Elementen initialisieren, die durch den Parameter vorgegeben ist

Schreiben Sie nun eine `main` Funktion, in der zunächst eine Listen-Variable angelegt wird. In das Listen Objekt sollen nun 10 Elemente eingefügt werden. Versuchen Sie nun ein 11. Element einzufügen und geben Sie die Liste dann aus.

Erweitern Sie die Klasse durch zwei Methoden

extend diese Methode soll das Feld, welches die Listenelemente enthält, um einen konstanten Faktor (z.B. um 10 Elemente) verlängern. Hierzu müssen Sie ein neues Feld von der aktuellen Länge + 10 erzeugen, die momentan im Feld enthaltenen Elemente kopieren und den Rest mit 0 auffüllen (und natürlich das Attribut, daß die Länge der Liste enthält, aktualisieren und den Speicher für das ursprüngliche Feld wieder freigeben). Definieren Sie das neue Feld wieder auf die oben beschriebene Art

insertElementAtEnd2 diese Methode soll prüfen, ob noch Platz in der Liste ist: wenn nicht, soll zuerst die Liste erweitert werden (mit der **extend**-Methode, bevor ein Element eingefügt wird.

Fügen Sie mit Hilfe der neuen Methode zwei weitere Elemente in die Liste ein.

Schreiben Sie nun eine weitere Methode **insertSorted**, die eine Element jeweils so einfügt, dass die Liste immer (aufsteigend) sortiert ist. Dazu müssen ab der Stelle, an der eingefügt werden muss, alle weiteren Elemente zunächst um eins nach hinten geschoben werden (Hinweis: fangen Sie hinten an).

Ergänzen Sie die main Funktion derart, dass 11 Elemente gelöscht werden (es sollte dann noch ein Element in der Liste sein) und fügen Sie dann mittels der neuen Methode 10 Elemente sortiert ein und geben Sie die Liste dann nochmals aus.

----- Liste.h -----

```
1 #ifndef LISTE_H_
2 #define LISTE_H_
3
4 class Liste
5 {
6     private:
7         int *elements;
8         unsigned int listLength;
9         unsigned int currentPosition;
10        void init();
11
12    public:
13        Liste ();
14        Liste (unsigned int numberOfElements);
15
16        void init(unsigned int i);
17        void insertElementAtEnd1(int element);
```

```

18 void insertElementAtEnd2(int element);
19 void extendList();
20
21 void insertSorted(int element);
22
23 void deleteElementFromEnd();
24
25 void printElementAt(unsigned int i);
26 void printList();
27 };
28
29 #endif // LISTE.H

```

Zeile 7-10: Deklariert die privaten Attribute und Methoden der Klasse

Zeile 13-26: Deklariert die öffentlichen Methoden der Klasse.

Was hier im Detail deklariert wird sollte mittlerweile klar sein.

----- Liste.cpp -----

```

1 #include "Liste.h"
2 #include <iostream>
3
4 Liste::Liste()
5 {
6     init(10);
7 }
8
9 Liste::Liste(unsigned int numElements)
10 {
11     init(numElements);
12 }
13
14 void Liste::init(unsigned int numElements)
15 {
16     if(numElements==0)
17     {
18         std::cout << "Number of elements in list should be greater than 0.";
19         std::cout << "Using 1." << "\n";
20         numElements=1;
21     }
22     elements = new int[numElements];
23     listLength = numElements;
24     unsigned int i;
25     for (i=0;i<numElements;i++)

```

```

26 {
27     elements[i] = 0;
28 }
29 currentPosition = 0;
30 }
31
32 void Liste::insertElementAtEnd1(int element)
33 {
34     if (currentPosition < listLength)
35     {
36         elements[currentPosition] = element;
37         currentPosition++;
38     }
39     else
40     {
41         std::cout << "list full: no more insertions possible" << "\n";
42     }
43 }
44
45 void Liste::insertElementAtEnd2(int element)
46 {
47     if (currentPosition > listLength)
48     {
49         extendList();
50     }
51     elements[currentPosition] = element;
52     currentPosition++;
53 }
54 }
55
56 void Liste::insertSorted(int element)
57 {
58     if (currentPosition > listLength)
59     {
60         extendList();
61     }
62
63     unsigned int pos = 0;
64     for (; (pos <= currentPosition) && (element > elements[pos]); ++pos)
65     {
66     }
67
68     if (pos < currentPosition)
69     {
70         unsigned int k;

```

```

71     for (k = currentPosition;k > pos; k--)
72     {
73         elements[k] = elements[k-1];
74     }
75
76     elements[pos] = element;
77     currentPosition++;
78 }
79 else
80 {
81     insertElementAtEnd1(element);
82 }
83 }
84
85
86 void Liste::extendList()
87 {
88     int *newElements;
89     newElements = new int[listLength + 10];
90     unsigned int i;
91     for (i=0;i<currentPosition;i++)
92     {
93         newElements[i] = elements[i];
94     }
95     listLength += 10;
96     for (i=currentPosition;i< listLength; i++)
97     {
98         newElements[i] = 0;
99     }
100 }
101 delete elements;
102 elements = newElements;
103 }
104
105 void Liste::deleteElementFromEnd()
106 {
107     if (currentPosition >= 0)
108     {
109         elements[currentPosition] = 0;
110         currentPosition--;
111     }
112     else
113     {
114         std::cout << "list is empty: no deletion possible" << "\n";
115     }

```

```

116 }
117
118 void Liste::printElementAt(unsigned int i)
119 {
120     if (i <= currentPosition)
121     {
122         std::cout << "element at position " << i << ": " << elements[i] << "\n";
123     }
124 }
125
126 void Liste::printList()
127 {
128     unsigned int i;
129     for (i = 0; i < currentPosition; i++)
130     {
131         printElementAt(i);
132     }
133 }

```

Zeile 4-7: Standardkonstruktor welcher über die *init* Methode die Attribute initialisiert und die Größe der Liste auf 10 Element setzt.

Zeile 9-12: Konstruktor welcher die gewünschte Größe der Liste als Argument entgegen nimmt und alle Attribute und die Größe mit Hilfe der *init* Methode setzt.

Zeile 14-30: Die *init* Methode überprüft ob die angegebenen Größe größer als Null ist, falls nicht wird eine Fehlermeldung ausgegeben und die Anzahl der Element auf eins gesetzt. Nun fordert die Methode mit **new[]** Speicher für die gewünschte Anzahl an Elementen an, initialisiert alle Werte der Liste mit Null und setzt die restlichen Attribute.

Zeile 32-43: Eine der zwei Methoden zum Einfügen von Daten an das Ende der Liste. Die Methode überprüft, ob die aktuelle Position in der Liste kleiner ist als das Ende. Wenn ja, dann wird die Zahl in die Liste eingefügt und das Positionsattribut wird um eins erhöht. Falls die Position bereits am Ende der Liste ist, wird lediglich eine Fehlermeldung ausgegeben. Die Zahl wird nicht in die Liste eingefügt.

Zeile 45-54: Diese Methode macht fast dasselbe wie die vorherige Methode. Der Unterschied besteht darin, dass, falls die Position bereits am

Ende der Liste steht, die Liste mit Hilfe der *extendList* Methode um 10 Elemente vergrößert und die Zahl anschließend eingefügt wird.

Zeile 56-83: Auch die *insertSorted* Methode fügt Zahlen in die Liste ein und vergrößert die Liste falls notwendig. Die Zahlen werden jedoch nicht am Ende angefügt, sondern so, dass die Liste die Zahlen nach ihrem Wert sortiert enthält. Dazu geht die Methode die Elemente der Liste der Reihe nach durch und sucht die erste Zahl, die größer ist als der Wert der eingefügt werden soll. Ist die Position gefunden, müssen zuerst alle Elemente von der gefundenen Position ab, um eine Stelle in der Liste nach hinten verschoben werden. Man beginnt damit am besten am hinteren Ende der Liste. Zum Schluss wird nun der neue Wert an der gefundenen Position eingefügt.

Zeile 86-103: Die Methode *extendList* vergrößert die Liste um 10 Elemente, dazu wird neuer Speicher angefordert, der für alle alten Daten plus 10 neue Element ausreicht. Anschließend werden die alten Daten in den neuen Speicher kopiert und der ursprüngliche Speicher mit `delete[]` freigegeben.

Zeile 105-116: Die Methode überprüft, ob die Liste überhaupt Elemente enthält, wenn ja, wird das letzte mit Null überschrieben und das Positionsattribute um eins erniedrigt. Ist die Liste leer, wird lediglich eine Fehlermeldung ausgegeben.

118-124: Gibt das Element mit Index *i* aus, falls *i* im gültigen Bereich liegt.

126-133: Gibt mit Hilfe der Methode *printElementAt* alle Elemente der Liste aus.

Auch hier gilt dasselbe wie in Aufgabe 1. Der angeforderte Speicher wird nicht in allen Fällen wieder freigegeben. Ein weiteres Problem dieser Listenklasse ist, dass die Liste zwar bei Bedarf erweitert werden kann, sie wird aber nicht wieder verkleinert wenn Elemente gelöscht werden. Man könnte sich z.B. vorstellen, dass die Klasse ihre Größe auch wieder verkleinert, wenn die tatsächliche Anzahl der Elemente in der Liste um mehr als 10 kleiner ist als die Größe der Liste.

----- main.cpp -----

```

1 #include "Liste.h"
2 #include <iostream>
3
4
5 int main()
6 {
7     Liste* l1=new Liste();
8     l1.insertElementAtEnd1(3);
9     l1.insertElementAtEnd1(5);
10    l1.insertElementAtEnd1(7);
11    l1.insertElementAtEnd1(2);
12    l1.insertElementAtEnd1(0);
13    l1.insertElementAtEnd1(8);
14    l1.insertElementAtEnd1(1);
15    l1.insertElementAtEnd1(29);
16    l1.insertElementAtEnd1(17);
17    l1.insertElementAtEnd1(37);
18    l1.insertElementAtEnd1(2);
19    l1.insertElementAtEnd2(29);
20    l1.insertElementAtEnd2(33);
21    l1.printList();
22    l1.deleteElementFromEnd();
23    l1.deleteElementFromEnd();
24    l1.deleteElementFromEnd();
25    l1.deleteElementFromEnd();
26    l1.deleteElementFromEnd();
27    l1.deleteElementFromEnd();
28    l1.deleteElementFromEnd();
29    l1.deleteElementFromEnd();
30    l1.deleteElementFromEnd();
31    l1.deleteElementFromEnd();
32    l1.deleteElementFromEnd();
33    std::cout << "-----" << "\n";
34    l1.printList();
35    std::cout << "-----" << "\n";
36    l1.insertSorted(27);
37    l1.insertSorted(73);
38    l1.insertSorted(2);
39    l1.insertSorted(3);
40    l1.insertSorted(1);
41    l1.insertSorted(99);
42    l1.insertSorted(98);
43    l1.insertSorted(97);
44    l1.insertSorted(95);
45    l1.insertSorted(50);

```

```
46  ll.printList ();
47  delete ll;
48  return 0;
49 }
```

Zeile 7: Deklariert einen Zeiger auf eine Liste und fordert Speicher für die Liste an. Die Liste, auf die der Zeiger zeigt, wird über den Standardkonstruktor initialisiert. In diesem Fall wäre es nicht unbedingt notwendig gewesen die Liste mit **new** anzulegen, aber es schadet auch nicht den Umgang mit **new** und **delete** etwas zu üben.

Zeile 8-18: Fügt über die Methode *insertElementAtEnd1* neue Zahlen in die Liste ein. Ist die Liste voll, gibt diese Methode eine Fehlermeldung aus und fügt die Zahl nicht ein.

Zeile 19-20: Fügt über die Methode *insertElementAtEnd2* neue Zahlen in die Liste ein. Ist die Liste voll, vergrößert diese Methode die Liste und fügt die Zahl anschließend ein.

Zeile 21: Gibt den Inhalt der Liste aus.

Zeile 22-32: Löscht jeweils das letzte Element der Liste.

Zeile 33-35: Gibt die Liste erneut aus.

Zeile 36-46: Fügt Zahlen sortiert in die Liste ein und gibt diese aus.

Zeile 47: Gibt den mit **new** angeforderten Speicher für die Liste wieder frei.