Übungsblatt 5

(18. Mai 2007)

(2 Punkte)

Aufgabe 1 Ergänzen Sie die obige Listenklasse um weitere Listenoperatoren.

- `A operator[](int i);` soll das i-te Element zurückgeben.
- `Li<A>* operator-(int i);` soll eine neue Liste erzeugen, indem von der Liste die ersten i Elemente weggelassen werden.
- `bool operator<(Li<A>* that);` soll die beiden Listen bezüglich ihrer Länge vergleichen.
- `bool operator==(Li<A>* that);` soll die Gleichheit auf Listen implementieren.

Testen Sie ihre Implementierung an einigen Beispielen.

Aufgabe 2 Schreiben Sie eine Funktion zur Funktionskomposition mit folgender Signatur:

```
1 template <typename a, typename b, typename c>
2 c composition(c f2(b),b f1(a),a x);
```

Sie sei spezifiziert durch die Gelichung: $composition(f_2, f_1, x) = f_2(f_1(x))$. Testen Sie Ihre Implementierung für verschiedene Typen.

Lösung

Ich gebe es zu. Eine schwere Aufgabe, aber eine kurze Lösung. Also viel zu lernen. Hier die erstaunlich einfache Lösung:

Compose.cpp

```
1 #include <string>
2 #include <iostream>
3 template <typename a, typename b, typename c>
4 c composition(c f2(b),b f1(a),a x){
5   return f2(f1(x));
6 }
```

Spannender ist natürlich das einmal auszutesten. Wir machen das mal mit drei Funktionen, auch unterschiedlicher Typen

Compose.cpp

```
7  int doppelt(int x){return 2*x;}
8  int square(int x){return x*x;}
9  int laenge(std::string x){return x.length();}
10
11 int main(){
12   std::string hallo="hallo";
13   std::cout<< composition(square,doppelt,2)      <<std::endl;
14   std::cout<< composition(doppelt,laenge,hallo)  <<std::endl;
15   std::cout<< composition(square,square,4)       <<std::endl;
16 }
```

Aufgabe 3 Schreiben Sie eine Funktion `fold` mit folgender Signatur:

Fold.h

```
1  template <typename a,typename b>
2  b fold(a xs [],int length,b op(b,a),b startV);
```

Die Spezifikation der Funktion sei durch folgende Gleichung gegeben:

$$\text{fold(xs,l,op,st)} = \text{op}(\ldots\text{op}(\text{op}(\text{op}(\text{st,xs[0]}),\text{xs[1]}),\text{xs[2]})\ldots,\text{xs[l-1]})$$

Oder bei Infixschreibweise der Funktion `op` gleichbedeutend über folgende Gleichung:

$$\text{fold}(\{x_0, x_1,,\ldots,x_{l-1}\},\text{l,op,st}) = st \text{ op } x_0 \text{ op } x_1 \text{ op } x_2 \text{ op } \ldots \text{op } x_{l-1}$$

Testen Sie Ihre Methode `fold` mit folgenden Programm:

FoldTest.cpp

```
1  #include <iostream>
2  #include <string>
3  #include "Fold.h"
4
5  int add(int x,int y){return x+y;}
6  int mult(int x,int y){return x*y;}
7  int gr(int x,int y){return x>y?x:y;}
8  int strLaenger(int x,std::string y){
```

```
9   return x>y.length()?x:y.length();}
10
11  int main(){
12    int xs [] = {1,2,3,4,5,4,3,2,1};
13    std::cout << fold(xs,9,add,0) << std::endl;
14    std::cout << fold(xs,9,mult,1)<< std::endl;
15    std::cout << fold(xs,9,gr,0)  << std::endl;
16    std::string ys [] = {"john","paul","george","ringo","stu"};
17    std::cout << fold(ys,5,strLaenger,0)  << std::endl;
18  }
```

Schreiben Sie ein paar zusätzliche Beispielaufrufe von `fold`.

Lösung

Auch diese Funktion ist eigentlich recht einfach, wenn man sie verstanden hat und aber richtig effektiv.

Fold.h

```
19  template <typename a,typename b>
20  b fold(a xs [],int length,b op(b,a),b startV){
21    b result = startV;
22    for (int i =0;i<length;i++) result=op(result,xs[i]);
23    return result;
24  }
```

Aufgabe 4 Sie sollen in dieser Aufgabe die Sortiermethode des Bubble-Sort, wie sie im ersten Semester und in ADS vorgestellt wurde, möglichst generisch umsetzen:

a) Implementieren Sie eine Funktion
`void bubbleSort(int xs [],int length )`,
die Elemente mit dem Bubblesort-Algorithmus der Größe nach sortiert. Schreiben Sie ein paar Tests für Ihre Funktion.

b) Schreiben Sie jetzt eine verallgemeinerte Sortierfunktion, in der die Sortierrelation als Parameter mitgegeben wird. Die Sortierfunktion soll also eine Funktion höherer Ordnung mit folgender Signatur sein:
`void bubbleSortBy(int xs [],int length, bool smaller(int,int) )`

c) Verallgemeinern Sie jetzt die Funktion `bubbleSortBy`, dass Sie generisch für Reihungen mit verschiedenen Elementtypen aufgerufen werden kann.

Lösung

Zunächst die Sortierung von ints nach ihrer Größe:

Bubble.cpp

```
#include <string>
#include <iostream>

void bubbleSort(int xs [],int l ){
  for (int i=l-1;i>=0;i--){
    for (int j=0;j<i;j++){
      if (xs[j]>xs[j+1]){
        int k=xs[j];
        xs[j]=xs[j+1];
        xs[j+1]=k;
      }
    }
  }
}
```

Nun so verallgemeinert, dass die Sortierrelation als Funktion übergeben wird:

Bubble.cpp

```
void bubbleSortBy(int xs [],int l, bool smaller(int,int) ){
  for (int i=l-1;i>=0;i--){
    for (int j=0;j<i;j++){
      if (smaller(xs[j+1],xs[j])){
        int k=xs[j];
        xs[j]=xs[j+1];
        xs[j+1]=k;
      }
    }
  }
}
```

Und schließlich weiter verallgemeinert, so dass der Typ der Elemente allgemein gehalten wurde:

Bubble.cpp

```
template <typename a>
void bubbleSortByGen(a xs [],int l, bool smaller(a,a) ){
  for (int i=l-1;i>=0;i--){
```

```
29      for (int j=0;j<i;j++){
30        if (smaller(xs[j+1],xs[j])){
31          a k=xs[j];
32          xs[j]=xs[j+1];
33          xs[j+1]=k;
34        }
35      }
36    }
37  }
```

Zum Abschluß ein paar Beispielaufrufe:

Bubble.cpp

```
38  bool smallerInt(int x, int y){return x<y;}
39
40  bool smallerString(std::string x, std::string y){
41    return x.length()<y.length();}
42
43  bool smallerString2(std::string x, std::string y){
44    return x<y;}
45
46  template <typename a>
47  void printArray(a xs [],int l){
48    std::cout<<"{";
49    for (int i=0;i<l-1;i++){
50      std::cout << xs[i]<<", ";
51    }
52    std::cout << xs[l-1] <<"}"<<std::endl;
53  }
54
55  int main(){
56    int xs1 [] = {32,4532,32,54,12,43,54,2};
57    printArray(xs1,8);
58    bubbleSort(xs1,8);
59    printArray(xs1,8);
60
61    int xs2 [] = {32,4532,32,54,12,43,54,2};
62    printArray(xs2,8);
63    bubbleSortBy(xs2,8,smallerInt);
64    printArray(xs2,8);
65
```

```
66    int xs3 [] = {32,4532,32,54,12,43,54,2};
67    printArray(xs3,8);
68    bubbleSortByGen(xs3,8,smallerInt);
69    printArray(xs3,8);
70
71    std::string xs4 []
72      = {"lear","macbeth","hamlet","othello","prospero"};
73    printArray(xs4,5);
74    bubbleSortByGen(xs4,5,smallerString);
75    printArray(xs4,5);
76
77    std::string xs5 []
78      = {"lear","macbeth","hamlet","othello","prospero"};
79    printArray(xs5,5);
80    bubbleSortByGen(xs5,5,smallerString2);
81    printArray(xs5,5);
82  }
```

Im besten Fall erkennt man aus den drei Sortierfunktionen, wie man sich aus einer speziellen Lösung nach und nach generellere Lösungen ableiten kann