Aufgabe 1

```
1  #include <iostream>
2
3  #include "SimpleArray.hh"
4  #include "SimpleArrayImp.cc"
5
6  template <class T>
7  class Polynomial :
8    public SimpleArray<T> {
9  public:
10   // konstruiere Polynom vom Grad n
11   Polynomial (int n);
12
13   // Default-Destruktor ist ok
14   // Default-Copy-Konstruktor ist ok
15   // Default-Zuweisung ist ok
16
17   // Grad des Polynoms
18   int degree ();
19
20   // Auswertung
21   T eval (T x);
22
23   // Addition von Polynomen
24   Polynomial operator+ (Polynomial q);
25
26   // Multiplikation von Polynomen
27   Polynomial operator* (Polynomial q);
28
29   // Gleichheit
30   bool operator== (Polynomial q);
31
32   // drucke Polynom
33   void print ();
34 } ;
35
36 template <class T>
37 Polynomial<T>::Polynomial (int n)
38   : SimpleArray<T> (n+1, 0) {}
39
40
41
42 // Grad auswerten
43 template <class T>
44 int Polynomial<T>::degree ()
```

```
45 {
46   return this->maxIndex();
47 }
48
49
50 // Addition von Polynomen
51 template <class T>
52 Polynomial<T> Polynomial<T>::operator+ (Polynomial<T> q)
53 {
54   int nr=degree(); // mein grad
55
56   if (q.degree()>nr) nr=q.degree();
57
58   Polynomial r(nr); // Ergebnispolynom
59
60   for (int i=0; i<=nr; i=i+1)
61     {
62       if (i<=degree())
63         r[i] = r[i]+(*this)[i]; // add me to r
64       if (i<=q.degree())
65         r[i] = r[i]+q[i];       // add q to r
66     }
67
68   return r;
69 }
70
71 // Multiplikation von Polynomen
72 template <class T>
73 Polynomial<T> Polynomial<T>::operator* (Polynomial<T> q)
74 {
75   Polynomial r(degree()+q.degree()); // Ergebnispolynom
76
77   for (int i=0; i<=degree(); i=i+1)
78     for (int j=0; j<=q.degree(); j=j+1)
79       r[i+j] = r[i+j] + (*this)[i]*q[j];
80
81   return r;
82 }
83
84 // Drucken
85 template <class T>
86 void Polynomial<T>::print ()
87 {
88   if (degree()<0)
89     std::cout << 0;
90   else
```

```
91      std::cout << (*this)[0];
92
93    for (int i=1; i<=this->maxIndex(); i=i+1)
94      std::cout << "+" << (*this)[i] << "*x^" << i;
95
96    std::cout << std::endl;
97  }
98
99  // Auswertung
100 template <class T>
101 T Polynomial<T>::eval (T x)
102 {
103   T sum=0;
104
105   // Hornerschema
106   for (int i=this->maxIndex(); i>=0; i=i-1)
107     sum = sum*x + (*this)[i];
108   return sum;
109 }
110
111 template <class T>
112 bool Polynomial<T>::operator== (Polynomial<T> q)
113 {
114   if (q.degree()>degree())
115     {
116       for (int i=0; i<=degree(); i=i+1)
117         if ((*this)[i]!=q[i]) return false;
118       for (int i=degree()+1; i<=q.degree(); i=i+1)
119         if (q[i]!=0) return false;
120     }
121   else
122     {
123       for (int i=0; i<=q.degree(); i=i+1)
124         if ((*this)[i]!=q[i]) return false;
125       for (int i=q.degree()+1; i<=degree(); i=i+1)
126         if ((*this)[i]!=0.0) return false;
127     }
128
129   return true;
130 }
131
132 int main()
133 {
134     Polynomial<float> p(2);
135     p[0] = 1.0;
136     p[1] = 1.0;
```

```
137     p.print();
138     p = p*p;
139     p.print();
140
141     Polynomial<double> q(3);
142     q[0] = 2.0;
143     q[1] = -1.0;
144     q[3] = 4.0;
145     q.print();
146 }
```

Aufgabe 2

```
1  #include <vector>
2  #include <iostream>
3
4  template <class T>
5  class Function
6  {
7  public:
8      virtual ~Function(){};
9      virtual T operator()(T &x) = 0;
10 };
11
12 template <class T, int n>
13 class Vector : public std::vector<T>
14 {
15 public:
16     Vector();
17     Vector(T entry);
18     //Addition zweier Vektoren
19     template <class T2>
20     Vector<T, n> operator+(const Vector<T2, n> &y);
21
22     //Skalarprodukt
23     template <class T2>
24     T operator*(const Vector<T2, n> &y);
25
26     //Skalarmultiplikation
27     template <class T2>
28     Vector<T, n> operator*(T2 y);
29
30     //Anwenden einer Funktion
31     void apply(Function<T> &func);
```

```
32
33      //minimum
34      T min();
35      //maximum
36      T max();
37      //mittelwert
38      T mittel();
39
40      void print();
41  };
42
43  template <class T, int n>
44  Vector<T, n>::Vector() : std::vector<T>(n) {}
45
46  template <class T, int n>
47  Vector<T, n>::Vector(T entry) : std::vector<T>(n, entry) {}
48
49  template <class T, int n>
50  template <class T2>
51  Vector<T, n> Vector<T, n>::operator+(const Vector<T2, n> &y)
52  {
53      Vector<T, n> ret;
54      for (size_t i = 0; i < n; i++)
55      {
56          ret[i] = (*this)[i] + y[i];
57      }
58      return ret;
59  }
60
61  template <class T, int n>
62  template <class T2>
63  T Vector<T, n>::operator*(const Vector<T2, n> &y)
64  {
65      T ret = 0;
66      for (size_t i = 0; i < n; i++)
67      {
68          ret += (*this)[i] * y[i];
69      }
70      return ret;
71  }
72
73  template <class T, int n>
74  template <class T2>
75  Vector<T, n> Vector<T, n>::operator*(T2 y)
76  {
77      Vector<T, n> ret;
```

```
78      for (size_t i = 0; i < n; i++)
79      {
80          ret[i] = (*this)[i] * y;
81      }
82      return ret;
83  }
84
85  template <class T, int n>
86  void Vector<T, n>::apply(Function<T> &func)
87  {
88      for (size_t i = 0; i < n; i++)
89      {
90          (*this)[i] = func((*this)[i]);
91      }
92  }
93
94  template <class T, int n>
95  T Vector<T, n>::min()
96  {
97      T ret = (*this)[0];
98      for (size_t i = 1; i < n; i++)
99      {
100         if ((*this)[i] < ret)
101         {
102             ret = (*this)[i];
103         }
104     }
105     return ret;
106 }
107
108 template <class T, int n>
109 T Vector<T, n>::max()
110 {
111     T ret = (*this)[0];
112     for (size_t i = 1; i < n; i++)
113     {
114         if ((*this)[i] > ret)
115         {
116             ret = (*this)[i];
117         }
118     }
119     return ret;
120 }
121
122 template <class T, int n>
123 T Vector<T, n>::mittel()
```

```
124 {
125     T sum = 0;
126     for (size_t i = 1; i < n; i++)
127     {
128         sum = sum + (*this)[i];
129     }
130     T ret = sum / n;
131     return ret;
132 }
133
134 template <class T, int n>
135 void Vector<T, n>::print()
136 {
137     std::cout << "[" << (*this)[0];
138     for (size_t i = 1; i < n; ++i)
139     {
140         std::cout << ", " << (*this)[i];
141     }
142     std::cout << "]";
143 }
144
145 int main()
146 {
147     Vector<double, 3> a(0.0);
148     a[0] = 1.0;
149     a[1] = 2.0;
150     a[2] = 3.0;
151     a.print();
152     std::cout << std::endl;
153     Vector<float, 3> b(0.0);
154     b[0] = 1.0;
155     b[1] = 2.0;
156     b[2] = 3.0;
157     b.print();
158     std::cout << std::endl;
159     float result = b * a;
160     a = a * result;
161     a.print();
162     class Quadrat : public Function<double>
163     {
164     public:
165         virtual double operator()(double &x) { return x * x; }
166     };
167     Quadrat quadrat = Quadrat();
168     a.apply(quadrat);
169     a.print();
```

```
170     a = a + b * -200;
171     a.print();
172     return 0;
173 }
```

Aufgabe 3

(a)

```
1  #include <iostream>
2  #include <cmath>
3
4  class Wurzel
5  {
6  public:
7      double operator()(double x)
8      {
9          return std::sqrt(x);
10     }
11 };
12
13 template <class Funktion>
14 double trapezregel(int n, double a, double b, Funktion& f)
15 {
16     double h = (b-a)/n;
17     double sum = 0;
18     for (size_t i = 1; i < n; i++)
19     {
20         sum += f(a + i*h);
21     }
22     return h/2 * (f(a) + 2 * sum + f(b));
23 }
24
25 int main()
26 {
27     Wurzel wurzel = Wurzel();
28     std::cout << trapezregel<Wurzel>(100, 0, 1.5, wurzel);
29     return 0;
30 }
```

(b)

- Vorteile: Es müssen keine Entscheidungen zur Laufzeit getroffen werden → Effizienz
- Nachteile: Kompilieren kann unter Umständen sehr lange dauern.