| Aufgabe | A1 | A2 | A3 | $\sum$ |
|---|---|---|---|---|
| Punkte | | | | |

**Aufgabe 1.** Polynomschablone. Die vorgegebene Implementation der `SimpleArray` Klasse ist in *simplearray.cc* zu finden. In *polynomial.cc* befindet sich die Implementation der verallgemeinerten `Polynomial` Klasse.

```
#include <iostream>
#include <stdio.h>
#include "simplearray.cc"

template <class T>
class Polynomial: public SimpleArray<T> {
    public:
        // konstruiere Polynom vom Grad n
        Polynomial<T> (int n);

        // Default-Destruktor ist ok
        // Default-Copy-Konstruktor ist ok
        // Default-Zuweisung ist ok

        // Grad des Polynoms
        int degree();

        // Auswertung
        T eval (T x);

        // Addition von Polynomen
        Polynomial<T> operator+ (Polynomial<T> q);

        // Multiplikation von Polynomen
        Polynomial<T> operator* (Polynomial<T> q);

        // Gleichheit
        bool operator== (Polynomial q);

        // drucke Polynom
        void print ();
};

// Constructor
template <class T>
Polynomial<T>::Polynomial(int n)
    : SimpleArray<T>::SimpleArray(n+1,0.0) {}

// Grad auswerten
template <class T>
int Polynomial<T>::degree ()
{
    return SimpleArray<T>::maxIndex();
}

// Addition von Polynomen
template <class T>
Polynomial<T> Polynomial<T>::operator+ (Polynomial<T> q) {
    int nr=degree(); // mein grad

    if (q.degree()>nr) nr=q.degree();

    Polynomial r(nr); // Ergebnispolynom

    for (int i=0; i<=nr; i=i+1)
    {
        if (i<=degree())
            r[i] = r[i]+(*this)[i]; // add me to r
        if (i<=q.degree())
            r[i] = r[i]+q[i];       // add q to r
    }

    return r;
}

// Multiplikation von Polynomen
template <class T>
```

```
68 Polynomial<T> Polynomial<T>::operator* (Polynomial<T> q)
69 {
70     Polynomial r(degree()+q.degree()); // Ergebnispolynom
71 
72     for (int i=0; i<=degree(); i=i+1)
73         for (int j=0; j<=q.degree(); j=j+1)
74             r[i+j] = r[i+j] + (*this)[i]*q[j];
75 
76     return r;
77 }
78 
79 // Drucken
80 template <class T>
81 void Polynomial<T>::print ()
82 {
83     if (degree()<0)
84         std::cout << 0;
85     else
86         std::cout << (*this)[0];
87 
88     for (int i=1; i<=SimpleArray<T>::maxIndex(); i=i+1)
89         std::cout << " + " << (*this)[i] << "*x^" << i;
90 
91     std::cout << std::endl;
92 }
93 
94 int main() {
95     Polynomial<float> p(2);
96     p[0] = 1.0;
97     p[1] = 1.0;
98     p.print();
99 
100     p = p*p;
101     p.print();
102 
103     Polynomial<double> q(3);
104     q[0] = 2.0;
105     q[1] = -1.0;
106     q[3] = 4.0;
107     q.print();
108 }
```

polynomial.cc

**Aufgabe 2.** Vektorimplementation

```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4  #include <numeric>
5 
6  // a class implementing a mathematical vector
7  template <class K, size_t N>
8  class Vector {
9      public:
10         // constructor, initializes vector with zeroes of length N
11         Vector() : _vector(N, 0) {}
12         // initialize by passing vector vec to initialize the vector object with
13         Vector(std::vector<K> vec) { _vector = vec; }
14 
15         // copy constructor, assignment, destructor not needed
16         // all handled by std::vector
17 
18         // reference [] operator, used for assignments
19         K& operator[](int i) {
20             return _vector[i];
21         }
22 
23         // const ref [] operator
24         K operator[](int i) const {
25             return _vector[i];
26         }
27 
28         // return an iterator pointing to the first element of the underlying std::
       vector
29         typename std::vector<K>::const_iterator begin() const {
30             return _vector.begin();
```

```
31          }
32
33          // return an iterator pointing to the last element of the underlying std::
        vector
34          typename std::vector<K>::const_iterator end() const {
35              return _vector.end();
36          }
37
38          // dot product of two vectors of the same length
39          template <class K2>
40          K operator*(const Vector<K2,N>& y) {
41              std::vector<K> result(N); // new empty std::vector of correct size
42              // now multiply one element of the first with one of the second vector
43              std::transform(_vector.begin(), _vector.end(), y.begin(), result.begin(),
        std::multiplies<K>());
44              // and sum up the products
45              return std::accumulate(result.begin(), result.end(), 0);
46          }
47
48          // vector addition
49          template <class K2>
50          Vector<K, N> operator+(const Vector<K2, N>& y) {
51              std::vector<K> result(N); // new empty std::vector of correct size
52              // add elements by adding one element of the first to one of the second
        vector
53              std::transform(_vector.begin(), _vector.end(), y.begin(), result.begin(),
        std::plus<K>());
54              return Vector(result);
55          }
56
57          // return the maximum value in the vector
58          K max() {
59              return *std::max_element(_vector.begin(), _vector.end());
60          }
61
62          // return the minimum value in the vector
63          K min() {
64              return *std::min_element(_vector.begin(), _vector.end());
65          }
66
67          // return the average value of the vector
68          K average() {
69              // sum up all elements
70              K sum = std::accumulate(_vector.begin(), _vector.end(), 0);
71              return sum / N; // and divide by length
72          }
73
74          // run the specified function on each vector element
75          template <class F>
76          Vector<K,N> map(F op) const {
77              std::vector<K> result(N); // initialize new std::vector of correct size
78              // run op on each element
79              std::transform(_vector.begin(), _vector.end(), result.begin(), op);
80              return Vector(result);
81          }
82      private:
83          std::vector<K> _vector; // the private std::vector container
84  };
85
86  // print vector
87  template <class K, size_t N>
88  std::ostream& operator<<(std::ostream& os, const Vector<K,N>& v) {
89      os << "["; // opening brackets
90      for (size_t i = 0; i < N; i++) {
91          os << v[i]; // print element
92          if(i < N-1) {
93              os << ", "; // if not the last one, add comma
94          }
95      }
96      os << "]"; // closing brackets
97      return os;
98  }
99
100 // scalar multiplication for vector * scalar
101 template <class K, size_t N, class K2>
102 Vector<K, N> operator*(const Vector<K,N>& v, const K2 k) {
```

```
103     // multiply a scalar value using the generic map function
104     return v.map(std::bind1st(std::multiplies<K>(), k));
105 }
106
107 // scalar multiplication for scalar * vector
108 template <class K, size_t N, class K2>
109 Vector<K, N> operator*(const K2 k, const Vector<K,N>& v) {
110     return v * k;
111 }
112
113
114 // example function
115 double foo(double val) {
116     return val*2+3;
117 }
118
119 // testing out functionalities
120 int main() {
121     Vector<double, 3> a;
122     Vector<float, 3> b;
123     a[0] = 0;
124     a[1] = 0;
125     a[2] = 4;
126
127     b[0] = 42;
128     b[1] = -20;
129     b[2] = 3;
130
131     std::cout << "Vektor a: " << a << ", Vektor b: " << b << std::endl;
132     std::cout << "a+b: " << a+b << std::endl;
133     std::cout << "a*b: " << a*b << std::endl;
134     std::cout << "42*b: " << 42*b << std::endl;
135     std::cout << "b*42: " << b*42 << std::endl;
136     std::cout << "max(a): " << a.max() << std::endl;
137     std::cout << "min(a): " << a.min() << std::endl;
138     std::cout << "mean(a): " << a.average() << std::endl;
139     std::cout << "foo angewendet auf b: " << b.map(foo) << std::endl;
140 }
```

vectors.cc

**Aufgabe 3** (Funktoren und statischer Polymorphismus)**.** (a) Integration mittels Trapezregel für $n = 10$, $n = 100$ und $n = 1000$ beispielsweise ausgerechnet.

```
1  #include <iostream>
2  #include <cmath>
3
4  // Implementiert die Trapezregel mit der Intervallanzahl n
5  // der unteren Integrationsgrenze a, der oberen Integrationsgrenze b
6  // und ueber die Funktion f
7  template <class Function>
8  double trapezregel(int n, double a, double b, Function f) {
9      double h = (b-a)/n; // berechne intervalllaenge
10     double accum = 0;
11     for (int i = 1; i < n; i++) {
12         accum += f(a + i*h); // fuehre die summe aus
13     }
14     return h/2 * (f(a) + 2*accum + f(b)); // wende formel an
15 }
16
17 // beispiel fuer funktor
18 class Wurzel {
19     public:
20         double operator()(double x) {
21             return std::sqrt(x);
22         }
23 };
24
25 int main() {
26     Wurzel f;
27     // fuehre integration fuer verschiedene Intervallanzahlen n aus
28     double integ10 = trapezregel(10, 0, 1.5, f);
29     double integ100 = trapezregel(100, 0, 1.5, f);
30     double integ1000 = trapezregel(1000, 0, 1.5, f);
31     std::cout << "Integral ueber sqrt(x) von 0 bis 1.5" << std::endl;
32     std::cout << "n = 10: " << integ10 << std::endl;
```

```
33      std::cout << "n = 100: " << integ100 << std::endl;
34      std::cout << "n = 1000: " << integ1000 << std::endl;
35  }
```

integration.cc

(b) Vorteile der Verwendung von statischem Polymorphismus gegenüber dynamischem Polymorphismus:

- Auswertung zum Zeitpunkt des Kompilierens führt zu besserer Optimierung und Fehlererkennung durch den Compiler.
- Weniger Overhead durch VTABLE's, etc.
- Deshalb effizienter und schneller

Nachteile:

- Potentiell unübersichtlicher
- Größe der Programme als kompilierte Binärdatei ist größer