Programming 2 - Assignment 5

Felix Dreßler (k12105003)
email FelixDressler01@gmail.com

June 2, 2022

```
1  #pragma once
2
3  #include<string>
4  #include<iostream>
5
6  using namespace std;
7
8  class Integer {
9  private:
10   int n;
11 public:
12   // integer with value n (default 0)
13   Integer(int n = 0);
14
15   // destructor - empty because in Integer no new arrays/pointers are created
16   ~Integer() {
17
18   }
19
20   // a heap-allocated duplicate of this element
21   Integer* clone();
22
23   // the string representation of this element
24   string str();
25
26   // the constant of the type of this element and the inverse of this element
27   Integer* zero();
28   Integer* operator-();
29
30   // sum and product of this element and c
31   Integer* operator+(Integer* c);
32   Integer* operator*(Integer* c);
33
34   // comparison function
35   bool operator==(Integer* c);
36 };
37
38
39 // integer with value n (default 0)
40 Integer::Integer(int n) {
41     this->n = n;
42 }
43
44 // a heap-allocated duplicate of this element
45 Integer* Integer::clone() {
46     Integer* c = new Integer(this->n);
47
48     return c;
49 }
50
51 // the string representation of this element
52 string Integer::str() {
53
54     return to_string(this->n);;
55 }
56
57 // the constant of the type of this element and the inverse of this element
58 Integer* Integer::zero() {
59
```

```
60      return new Integer(0);
61  }
62
63  Integer* Integer::operator-() {
64
65      return new Integer(-(this->n));
66  }
67
68  // sum and product of this element and c
69  Integer* Integer::operator+(Integer* x) {
70      //Integer* x = dynamic_cast<Integer*>(c);
71
72      ////if cast is unsuccessful, we exit
73      //if (x == 0) {
74      //    cout << "Error: Addition with incompatible Elements performed" << endl;
75      //    exit(1);
76      //}
77
78      int t = this->n + x->n;
79
80      return new Integer(t);
81  }
82
83  Integer* Integer::operator*(Integer* x) {
84      //Integer* x = dynamic_cast<Integer*>(c);
85
86      ////if the cast is unsuccessful, we exit
87      //if (x == 0) {
88      //    cout << "Error: Multiplikation with incompatible Elements performed" << endl;
89      //    exit(2);
90      //}
91
92      return new Integer(this->n * x->n);
93  }
94
95  // comparison function
96  bool Integer::operator==(Integer* x) {
97      //Integer* x = dynamic_cast<Integer*>(c);
98
99
100     ////if cast is unsuccessful, we exit
101     //if (x == 0) {
102     //    cout << "Error: Comparison with incompatible Elements performed" << endl;
103     //    exit(3);
104     //}
105
106     if (this->n == x->n) {
107         return true;
108     }
109     else {
110         return false;
111     }
112
113 }
```

```
1  #pragma once
2
3  #include<string>
4  #include<iostream>
```

```
5
6  #include"Integer.h"
7
8
9  using namespace std;
10
11 template<class Ring> class RecPoly
12 {
13 private:
14   Ring** coeff;
15   int n;
16   string var;
17
18 public:
19
20     // polynomial with n>=0 coefficients and given variable name
21     RecPoly(string var, int n, Ring** coeffs) {
22         this->var = var;
23
24         int zeros = 0;
25
26         //cuts of all 0s at the end of the coeffs array
27         if (n != 0) {
28             Ring* z = coeffs[0]->zero();
29
30             for (int i = n - 1; i >= 0; i--) {
31                 if (!(coeffs[i]->operator==(z))) {
32                     break;
33                 }
34                 zeros++;
35             }
36             delete z;
37         }
38
39         this->n = n - zeros;
40         this->coeff = new Ring * [n];
41         for (int i = 0; i < n; i++) {
42             coeff[i] = coeffs[i]->clone();//clone to make sure only we have control over
                the array
43         }
44     }
45
46     // copy constructor, copy assignment operator
47     RecPoly(RecPoly& p) {
48         this->var = p.var;
49         this->n = p.n;
50
51         this->coeff = new Ring * [n];
52         for (int i = 0; i < n; i++) {
53             coeff[i] = p.coeff[i]->clone();
54         }
55     }
56
57     RecPoly& operator=(RecPoly& p) {
58         this->var = p.var;
59         this->n = p.n;
60
61         delete[] this->coeff;
62         this->coeff = new Ring * [n];
63
```

```
64          for (int i = 0; i < n; i++) {
65              coeff[i] = p.coeff[i]->clone();
66          }
67
68          return *this;
69      }
70
71      //destructor for RecPoly
72      ~RecPoly() {
73          for (int i = 0; i < this->n; i++) {
74              delete coeff[i];
75          }
76          delete[] coeff;
77      }
78
79      // a heap-allocated duplicate of this element
80      RecPoly* clone() {
81
82          return new RecPoly(*this);
83      }
84
85      // the string representation of this element
86      string str() {
87          string str = "";
88          if (n == 0) {
89              str = "0";
90          }
91          else {
92              str += "(";
93              for (int i = 0; i < n; i++) {
94                  if (!(coeff[i]->operator==(coeff[i]->zero()))) {
95                      str += coeff[i]->str();
96                      if (i != 0) {
97                          str += "*" + var + "^" + to_string(i);
98                      }
99                      if (i < n - 1) {
100                         str += "+";
101                     }
102                 }
103             }
104             str += ")";
105         }
106
107         return str;
108     }
109
110     // the constant of the type of this element and the inverse of this element
111     RecPoly* zero() {
112         return new RecPoly(this->var, 0, {});
113     }
114
115     RecPoly* operator-() {
116
117         Ring** temp = new Ring * [this->n];
118
119         for (int i = 0; i < this->n; i++) {
120             temp[i] = this->coeff[i]->operator-();
121         }
122
123         RecPoly* ret = new RecPoly(this->var, this->n, temp);
```

```
124
125         for (int i = 0; i < this->n; i++) {
126             delete temp[i];
127         }
128         delete[] temp;
129 
130         return ret;
131     }
132 
133     // sum operator for polynomials
134     RecPoly* operator+(RecPoly* x) {
135 
136        // RecPoly* x = dynamic_cast<RecPoly*>(c);
137 
138         //if (x == 0) {
139         //    cout << "Error: Addition with incompatible Elements performed" << endl;
140         //    exit(3);
141         //}
142         if (this->var != x->var) {
143             cout << "Error: Addition with incompatible Polynomials performed (wrong
                    variables)" << endl;
144             exit(4);
145         }
146 
147         else {
148             int n_temp = max(this->n, x->n);
149 
150             Ring** temp = new Ring * [n_temp];
151 
152             if (this->n == 0) {
153                 for (int i = 0; i < x->n; i++) {
154                     temp[i] = x->coeff[i]->clone();
155                 }
156 
157                 RecPoly* add = new RecPoly(this->var, x->n, temp);
158 
159                 for (int i = 0; i < n_temp; i++) {
160                     delete temp[i];
161                 }
162                 delete[] temp;
163 
164                 return add;
165             }
166             else {
167 
168                 for (int i = 0; i < this->n && i < x->n; i++) {
169                     temp[i] = this->coeff[i]->operator+(x->coeff[i]);
170                 }
171 
172                 if (this->n > x->n) {
173                     for (int i = x->n; i < this->n; i++) {
174                         temp[i] = this->coeff[i];
175                     }
176                 }
177                 else if (this->n < x->n) {
178                     for (int i = this->n; i < x->n; i++) {
179                         temp[i] = x->coeff[i];
180                     }
181                 }
182 
```

```
183                 RecPoly* add = new RecPoly(this->var, n_temp, temp);
184
185                 //for (int i = 0; i < n_temp; i++) {
186                 //    delete temp[i];
187                 //}
188                 //delete[] temp;
189
190                 return add;
191             }
192         }
193     }
194
195
196     // multiplication operator for polynomials
197     RecPoly* operator*(RecPoly* x) {
198
199         //RecPoly* x = dynamic_cast<RecPoly*>(c);
200
201         //if (x == 0) {
202         //    cout << "Error: Multiplication with incompatible Elements performed" <<
                endl;
203         //    exit(5);
204         //}
205         if (this->var != x->var) {
206             cout << "Error: Multiplication with incompatible Polynomials performed (wrong
                    variables)" << endl;
207             exit(6);
208         }
209
210         else {
211             if (this->n == 0 || x->n == 0) {
212                 return this->zero(); // new RecPoly(this->var, 0, {});
213             }
214             else {
215
216                 int length = this->n + x->n - 1;
217
218                 Ring** temp = new Ring * [length];
219
220                 for (int i = 0; i < length; i++) {
221                     temp[i] = x->coeff[0]->zero();
222                 }
223
224                 for (int i = 0; i < this->n; i++) {
225                     for (int j = 0; j < x->n; j++) {
226                         Ring* del = temp[i + j];
227                         temp[i + j] = temp[i + j]->operator+(this->coeff[i]->operator*(x
                                ->coeff[j]));
228                         delete del;
229                     }
230
231                 }
232
233                 RecPoly* mult = new  RecPoly(this->var, length, temp);
234
235                 for (int i = 0; i < length; i++) {
236                     delete temp[i];
237                 }
238                 delete[] temp;
239
```

```
240                 return mult;
241             }
242         }
243 
244     }
245 
246     // comparison function for Plynomials
247     bool operator==(RecPoly* x) {
248 
249         //RecPoly* x = dynamic_cast<RecPoly*>(c);
250 
251         bool same = true;
252 
253         for (int i = 0; i < this->n; i++) {
254             if (this->coeff[i] != x->coeff[i]) {
255                 same = false;
256             }
257         }
258         return same;
259     }
260 
261 
262   //// polynomial with n>=0 coefficients and given variable name
263   //RecPoly(string var, int n, Ring** coeffs);
264   //// copy constructor, copy assignment operator, destructor
265   //RecPoly(RecPoly& p);
266   //RecPoly& operator=(RecPoly& p);
267   //~RecPoly();
268 
269   ////functions from Ring:
270 
271   //// a heap-allocated duplicate of this element
272   //Ring* clone();
273 
274   //// the string representation of this element
275   //string str();
276 
277   //// the constant of the type of this element and the inverse of this element
278   //Ring* zero();
279   //Ring* operator-();
280 
281   //// sum and product of this element and c
282   //Ring* operator+(Ring* c);
283   //Ring* operator*(Ring* c);
284 
285   //// comparison function
286   //bool operator==(Ring* c);
287 };
288 
289 typedef RecPoly<Integer> UniPoly;
290 
291 typedef RecPoly<UniPoly> BiPoly;
```

```
1 //*************************************************************
2 // "Main.cpp"
3 //
4 // is used to test the functions of Ring and its subclasses.
5 //
6 // created by: Felix Dressler - 24.05.2022
```

```
7  //*************************************************************
8
9  #include "Integer.h"
10 #include "RecPoly.h"
11 #include <iostream>
12
13 using namespace std;
14
15 int main() {
16     //tests from the assignment
17     Integer* c[] = { new Integer(-5), new Integer(2), new Integer(0), new Integer(-3) };
18
19     UniPoly* p = new UniPoly("x", 4, c); // p = -3x^3 + 2x - 5
20     cout << p->str() << endl;
21
22     UniPoly* q = // q = p+p = -6x^3 + 4x - 10
23         p->operator+(p);
24     cout << q->str() << endl;
25
26     UniPoly* r = // r = p*q = 50 - 40 x + 8 x^2 + 60 x^3 - 24 x^4 + 18 x^6
27         p->operator*(q);
28     cout << r->str() << endl;
29
30     //additional tests
31
32     //zero
33
34     UniPoly* z = //zero polynomial
35         p->zero();
36     cout << z->str() << endl;
37
38     UniPoly* z2 = // = p + 0
39         p->operator+(z);
40     cout << z2->str() << endl;
41
42     UniPoly* z3 = // = p * 0
43         p->operator*(z);
44     cout << z3->str() << endl;
45
46     //negation
47
48     UniPoly* g = //= -p
49         p->operator-();
50     cout << g->str() << endl;
51
52     UniPoly* g2 = // = -p + p = 0
53         g->operator+(p);
54     cout << g2->str() << endl;
55
56     //multivariate polynomials
57
58     UniPoly* k[] = { p,q };
59     UniPoly* l[] = { p };
60
61     BiPoly* s = new BiPoly("y", 2, k);
62     cout << s->str() << endl;
63
64     BiPoly* s2 = new BiPoly("y", 1, l);
65     cout << s2->str() << endl;
66
```

```
67      BiPoly* s3 = // s2 + s
68          s2->operator+(s);
69      cout << s3->str() << endl;
70
71      BiPoly* s4 = //s2*s
72          s2->operator*(s);
73      cout << s4->str() << endl;
74
75      return 0;
76  }
```