Computer Systems (SS 2018)
Exercise 4: May 24, 2018

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC)
Wolfgang.Schreiner@risc.jku.at

May 8, 2018

The exercise is to be submitted by the denoted deadline via the submission interface of the Moodle course as a single file in zip (`.zip`) or tarred gzip (`.tgz`) format which contains the following files:

- A PDF file `Exercise`*`Number-MatNr`*`.pdf` (where *Number* is the number of the exercise and *MatNr* is your "Matrikelnummer") which consists of the following parts:
  1. A decent cover page with the title of the course, the number of the exercise, and the author of the solution (identified by name, Matrikelnummer and email address).
  2. For every source file, a listing in a *fixed width font*, e.g. `Courier`, (such that indentations are appropriately preserved) and an appropriate *font size* such that source code lines do not break.
  3. A description of all tests performed (copies of program inputs and program outputs) explicitly highlighting, if some test produces an unexpected result.
  4. Any additional explanation you would like to give. In particular, if your solution has unwanted problems or bugs, please document these explicitly (you will get more credit for such solutions).
- Each source file of your solution (no object files or executables).

Please obey the coding style recommendations posted on the course site.

Exercise 4: Generic Polynomials by Inheritance

A univariate polynomial $c_n \cdot x^n + c_{n-1} \cdot x^{n-1} + \ldots + c_1 \cdot x + c_0 = \sum_{i=0}^{n} c_i \cdot x^i$ of degree n can be essentially represented by an array of its $n+1$ coefficients $c_0, \ldots, c_n$. The goal of this exercise is to implement a generic polynomia class `Polynomial` of univariate polynomials whose coefficients may be from any type that supports the usual ring operations. Based on this type, we will build univariate polynomials of type $\mathbb{Z}[x]$ (coefficients are from $\mathbb{Z}$) but also bivariate polynomials of type $\mathbb{Z}[x, y]$ by identifying this type with the univariate polynomial type $\mathbb{Z}[x][y]$ (coeffients are from $\mathbb{Z}[x]$); this is also called the *recursive* representation of multivariate polynomials.

In more detail, the implementation shall work as follows:

1. Take the following abstract class `Ring`:

   ```
   class Ring {
   public:
     // destructor
     virtual ~Ring() {}

     // a heap-allocated duplicate of this element
     virtual Ring* copy() = 0;

     // the string representation of this element
     virtual string str() = 0;

     // the constant of the type of this element and the inverse of this element
     virtual Ring* zero()  = 0;
     virtual Ring* operator-() = 0;

     // sum and product of this element and c
     virtual Ring* operator+(Ring* c) = 0;
     virtual Ring* operator*(Ring* c) = 0;

     // comparison function
     virtual bool operator==(Ring* c) = 0;
   };
   ```

2. Implement a concrete class `Integer`

   ```
   class Integer: public Ring {
   public:
     // integer with value n (default 0)
     Integer(long n=0);
   };
   ```

   This class overrides all the abstract (pure virtual) operations of class `Ring` by concrete definitions for integer arithmetic (where integers are represented by `long` values).

   Note that in the definition of the arithmetic and comparison functions the parameter `c` must be explicitly converted from type `Ring*` to type `Integer*`. Use the expression `dynamic_cast<Integer*>(c)` to receive a pointer to a `Integer` object (respectively `0`, if the conversion is not possible; the program may then be aborted with an error message).

3. Implement a concrete class `Polynomial`

```
class Polynomial: public Ring {
public:
  // polynomial with n>=0 coefficients and given variable name
  Polynomial(string var, int n, Ring **coeffs);

  // destructor
  virtual ~Polynomial();
};
```

   which implements univariate polynomials with generic coefficient types (i.e. coefficients that are represented by a concrete subclass of class `Ring`).

   The class stores internally an array of the polynomial coefficients as `Ring*` values (i.e., pointers to objects of (a subclass of) class `Ring`) where the leading coefficient is not zero (the zero polynomial is represented by an array of length zero). The constructor thus ignores trailing zeros in the given arrays. The coefficient array is to be allocated on the heap and filled with the duplicate *r*`.copy()` of every `Ring` element *r* of the given array; the destructor of the class thus frees these elements and the array. Please note that the constructor does neither store the array passed as an argument nor its elements!

   Class `Polynomial` is itself a concrete subclass of `Ring`; it thus overrides the abstract (pure virtual) operations of class `Ring` by concrete definitions for polynomial arithmetic. The class also overrides the virtual destructor of class `Ring` to free the array that was allocated when the polynomial was constructed and its elements. The text representation *p*`.str()` of a polynomial *p* shall be surrounded by parentheses `(...)` to also allow for polynomials with polynomial coefficients.

   When adding/multiplying two polynomials, first allocate a temporary `Ring*` array for the coefficients of the result polynomial and then fill this array with the appropriate coefficients; finally construct the result polynomial from this array using the constructor and free the array. Do not forget to free temporary coefficient values that you have created but that are not needed any more.

   Please note that class `Polynomial` does *not* use the class `Integer` described above!

Class `Polynomial` can be now tested with univariate polynomials:

```
Ring* c[] = { new Integer(-5), new Integer(2),
              new Integer(0),  new Integer(-3) };
Polynomial* p = new Polynomial("x", 4, c);        // p = -3x^3 + 2x - 5
cout << p->str();
Polynomial* q =                                   // q = p+p
  dynamic_cast<Polynomial*>(p->operator+(p));
cout << q->str();
Polynomial* r =                                   // r = p*q
```

```
    dynamic_cast<Polynomial*>(p->operator*(q));
  cout << r->str();
```

However, it also works with bivariate polynomials:

```
  Ring* d[] = { p, q, r };
  Polynomial* s = new Polynomial("y", 3, d);         // s = r*y^2 + q*y + p
  cout << s->str();
  Polynomial* t =                                    // t = s+s
    dynamic_cast<Polynomial*>(s->operator+(s));
  cout << t->str();
  Polynomial* u =                                    // u = s*t;
    dynamit_cast<Polynomial*>(s->operator*(t));
  cout << u->str();
```

Test class `Polynomial` in a *comprehensive* way (several calls of each function including the calls above); print the function results and show the outputs.