Computer Systems (SS 2016)
Exercise 4: May 24, 2016

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

May 13, 2016

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 $\sum_{i=0}^{n} c_i \cdot x^i$ of degree n can be represented by an array of its $n+1$ coefficients $c_0, \ldots, c_n$. The goal of this exercise is to implement in this way a class `PolyRat` of univariate polynomials with rational coefficients. The implementation shall be based on a generic polynomial class `Polynomial` which works for arbitrary coefficient types that support the usual ring operations.

In more detail, the implementation shall work as follows:

1. Take the following abstract class `Ring`:

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

  // 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+(const Ring* c) = 0;
  virtual Ring* operator*(const Ring* c) = 0;

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

2. Implement a concrete class `Rational`

```
class Rational: public Ring {
public:
  // rational with numerator n and denominator d (default 0/1)
  Rational(int n=0, int d=1);
};
```

   This class overrides all the abstract (pure virtual) operations of class `Ring` by concrete definitions for rational arithmetic (see Exercise 2, but you may use in this exercise plain `int` values for the numerators and the denominators).

   Note that in the definition of the arithmetic and comparison functions the parameter `c` must be explicitly converted from type `const Ring*` to type `const Rational*`. Use `dynamic_cast<const Rational*>(c)` to receive a pointer to a `Rational` 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 degree+1 coefficients and given variable name
  Polynomial(string var, int degree, Ring **coeffs);

  // destructor
  virtual ~Polynomial();

  // evaluate this polynomial on c
  const Ring* eval(const Ring *c) const;
};
```

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; the destructor of the class thus frees this array. Please note that the constructor does not store the array passed as an argument as the result array!

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.

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.

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

4. Derive from `Polynomial` the concrete class `PolyRat` whose objects denote polynomials with rational number coefficients. The interface of this class supports (by inheritance) the same operations as those of `Polynomial` but also provides a constructor

   ```
   PolyRat(string var, int degree, Rational **coeffs)
   ```

   This constructor creates by a declaration `Ring* c[degree+1];` an array of `Ring*` pointers into which the pointers of `coeffs` are copied; this array is then passed to the constructor of the parent class to initialize the polynomial (this auxiliary array is needed, because a `Rational**` value cannot be converted to type `Ring**`, even if a `Rational*` value can be converted to type `Ring*`).

   Please note that only this constructor needs to be implemented, all other operations are just inherited!

The classes thus support the following operations:

```
Rational* c[] = { new Rational(-5,2), new Rational(2,3),
                  new Rational(0,1),  new Rational(-3,-6) };
PolyRat p("x", 3, c);
cout << p.str();            // 1/2 x^3 + 2/3 x - 5/2
Rational r(1,2);
const Rational* s =
  dynamic_cast<const Rational*>(p.eval(r)); // evaluate for x = 1/2
cout << s->str();
const Polynomial *q =
  dynamic_cast<const Polynomial*>(p+(&p));
cout << q->str();
const Polynomial *t =
  dynamic_cast<const Polynomial*>(p*q);
cout << t->str();
```

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