Computer Systems (SS 2015)
Exercise 4: May 18, 2015

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

April 24, 2015

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

The goal of this exercise is to implement 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() const = 0;

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

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

  // comparison function
  virtual bool operator==(const Ring* c) const = 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.

   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 implementation proceeds in analogy to the the class `Polynomial` of Exercise 2 except that the internal array holds `Ring*` values,i.e., pointers to objects of (a subclass of) class `Ring`.

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.

Please note that in this exercise (unlike Exercise 2), class `Polynomial` does *not* know and 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 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 the constructor needs to be implemented, all other operations are just inherited!

5. Also derive from `Polynomial` the concrete class `PolyRat2` whose objects denote bivariate polynomials with rational number coefficients (in recursive representation). The interface of this class provides a constructor

   ```
   PolyRat2(string var, int degree, PolyRat **coeffs)
   ```

   Please note again that only the constructor has to be implemented!

   Hint: please override in class `Polynomial` the operation `str()` such that the text representation of a polynomial is surrounded by parentheses `()`; an object of type `PolyRat` is then printed like in

   ```
   ((3*y+1)*x^2+(2*y^2+5*y)*x+(2*y^4)*3)
   ```

Test classes `Rational`, `PolyRat`, and `PolyRat2` in a *comprehensive* way (several calls of each function) in a similar way as in Exercise 2 (print the function results and show the output).