Computer Systems (SS 2013)
Exercise 4: May 20, 2013

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

April 16, 2013

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

A univariate polynomial $\sum_{i=0}^{n} c_i \cdot x^i$ of degree n can be represented by its $n+1$ coefficients $c_0, \ldots, c_n$. The goal of this exercise is to implement a corresponding class `Poly` whose objects represent univariate polynomials with rational coefficients. The implementation shall be based on a generic polynomial class 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 generic polynomial class `GPoly` with the following interface:

```
class GPoly {
public:
  // constructor (constant polynomial)
  GPoly(const Ring* c);

  // adds coefficient, increases degree
  void addCoefficient(const Ring* c);

  // destructor
  virtual ~GPoly() {}

  // string representation using x for the variable
  // (don't show zero coefficients)
  virtual string str(const string &x) const;

  // evaluate polynomial on given value
```

```
  const Ring* eval(const Ring* c) const;
};
```

The class stores internally an array of pointers to the polynomial coefficients; initially the array holds one element (denoting a constant polynomial) to which subsequently further coefficients are added (starting with the low powers); the array is to be resized on demand.

3. Derive from `Ring` a concrete class `Rational`; every object of this class encapsulates a rational number represented by the numerator and denominator (a pair of `int` values that are relatively prime such that the nominator is greater than zero, use the Euclidean algorithm to divide common factors). The derived class provides implementations for all abstract operations of the base class (and possibly some extra operations).

   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 the corresponding `Rational` object (respectively `0`, if the conversion is not possible; the program may then be aborted with an error message).

4. Derive from `GPoly` a concrete class `Poly` whose components denote rational numbers. The interface of this class shall support the following operations:

```
int degree = 3;                          // length of arrays minus one
int numerators[]   = { -5, 2, 0, -3 };
int denumerators[] = {  2, 3, 1, -6 };
Poly p(degree, numerators, denumerators);
cout << p.str("x");                      // 1/2 x^3 + 2/3 x - 5/2
Rational* r = p.evaluate(1, 2);          // evaluate for x = 1/2
cout << r.str();
```

   The core idea of this implementation is to derive `Poly` from `GPoly` and use `Rational` as the coefficient domain. Thus class `Poly` inherits the data representation from `GPoly` (no new representation is required). Furthermore, some operations (e.g. `str()`) can be inherited without change, some operations have to be defined but can call in their definition the corresponding operations of the superclass (e.g. the constructor and `evaluate()`).

Test classes `Rational` and `Poly` in a *comprehensive* way (several calls of each function) including also the calls shown above (print the function results and show the output in the deliverable).