Computer Systems (SS 2013)
Exercise 6: June 17, 2013

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

May 23, 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 to 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 6: Polynomials by Sequence Containers

Take the following interface for a univariate polynomial over a coefficient domain `Ring`:

```
template<typename Ring> class Polynomial {
public:
  // destructor
  virtual ~Polynomial() {}

  // string representation using x for the variable
  // (don't show zero coefficients)
  virtual string str(const string& x) const = 0;

  // evaluate polynomial on given value
  virtual const Ring eval(const Ring& c) const = 0;
};
```

We assume that class `Ring` is specified as in Exercise 5 except that this class also provides public member functions

```
Ring(int d) { ...; }              // convert int to Ring
operator double() { return ...; } // convert Ring to double
```

The `operator double()` is a *conversion function* which returns a numerical approximation of the Ring value; every conversion `static_cast<double>(c)` of Ring value c to `double` automatically invokes this function.

Derive from `Polynomial` a non-abstract template class

```
template<typename Ring>
class DensePolynomial: public Polynomial<Ring> {...}
```

with constructor

```
// create polynomial of denoted degree with denoted coefficients
// 'coeffs' is a coefficient array of length 'degree+1'
// where 'coeffs[i]' represents the coefficient of exponent 'i'
DensePolynomial(int degree, Ring* coeffs)
```

that represents a polynomial by the sequence of all coefficients. This sequence is to be implemented with the help of the standard library as an object of type `vector<Ring>`; the implementation shall as far as possible make use of the operations that are already available on this type.

Also derive from `Polynomial` a non-abstract template class

```
template<typename Ring>
class SparsePolynomial: public Polynomial<Ring> {...}
```

with constructor

```
// create a polynomial from a sequence of 'number' monomials
// each represented by an exponent 'exps[i]' and coefficient 'coeffs[i]'
// 'exps' is in strict increasing order, 'coeffs' does not contain zeros
SparsePolynomial(int number, int* exps, Ring* coeffs)
```

that represents a polynomial by the sequence of all monomials with non-zero coefficients. The sequence is to be implemented with the help of the standard library as an object of type `list<Mono>` (where `Mono` is a user-defined monomial type containing the exponent and the coefficient of the monomial); the implementation shall as far as possible make use of the operations that are already available on this type.

Please note that by the use of the standard library classes it is not necessary to explicitly allocate heap memory with the operator `new`; thus there is also no need to re-define the default copy constructors, copy assignment operators, and destructors.

Finally implement the generic function

```
// draw polynomial p in interval [from,to] where
// x,y denotes the position of the coordinate center and
// s is a factor by which the polynomial values are to be scaled (multiplied)
template<typename Ring>
void draw(const Polynomial<Ring> &p, int from, int to, int x, int y, double s);
```

Use this function and the object function `str()` for printing a polynomial to test above classes *in an extensive way* on polynomials of type `DensePolynomial<Rational>` and `SparsePolynomial<Rational>` where `Rational` is a class analogous to that in Exercise 5 that may stand for `Ring`.