Computer Systems (SS 2012)
Exercise 4: May 21, 2012

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

May 4, 2012

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 $n+1$ coefficients $c_0, \ldots, c_n$. The goal of this exercise is to implement a corresponding class `Poly` whose objects represent univariate polynomials with double precision floating point 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 `Coeff`:

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

  // prints coefficient on standard output stream
  virtual void print() const = 0;

  // pointer to sum, and product
  virtual const Coeff* operator+(const Coeff* c) const = 0;
  virtual const Coeff* operator*(const Coeff* c) const = 0;
};
```

2. Implement a generic polynomial class `GPoly` with the following interface:

```
class GPoly {
public:
  // constructor (empty polynomial, not yet valid)
  GPoly();

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

  // destructor
  virtual ~GPoly() {}

  // degree of polynomial
  int degree() const;

  // coefficient of monomial with power i (<= degree)
  const Coeff* operator[](int i) const;

  // evaluate polynomial on given value
  const Coeff* eval(const Coeff* c) const;

  // print polynomial using  the given string for the variable
  virtual void print(char *var) const;
};
```

The class stores internally an array of pointers to the polynomial coefficients; initially the array is empty (denoting an invalid polynomial) to which subsequently the coefficients are added (starting with the low powers).

3. Derive from `Coeff` a concrete class `Double`; every object of this class encapsulates a double precision floating point number. 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 Coeff*` to type `const Double*`. Use `dynamic_cast<const Double*>(c)` to receive a pointer to the corresponding `Double` object (respectively `NULL`, 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 double precision floating point numbers. The interface of this class shall support the following operations:

   ```
   unsigned int degree = 3; double coeffs[] = { 1.5, -1, 0, 1 };
   Poly p(degree, coeffs);       // 1.5x^3 - x^2 + 1
   int n = p.degree();           // 3
   double coeff = p[n];          // 1.5
   double value = p.eval(1.0);   // 1.5
   p.print("y");                 // "1.5y^3 - y^2 + 1"
   p.draw(-10, 10, 150, 150, 5, 0.1);
   ```

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

   The function `draw` shall provide the following interface:

   ```
   // draws polynomial in interval [from, to] where
   // x,y denotes the position of the coordinate center and
   // h,v denote horizontal/vertical scaling factors
   void draw(int from, int to, int x, int y, double h, double v) const;
   ```

   This function evaluates the polynomial on all integers in the given interval, scales and shifts the result points as denoted by the given parameters, and draws them (connected by straight lines) on the screen.

Test class `Poly` in a comprehensive way including also the calls shown above (print the function results).