Object-Oriented Programming in C++ (SS 2019)
Exercise 4: May 23, 2019

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

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

Write a program that implements a type `Polynomial` of multivariate polynomials in distributive representation: here a polynomial $p \in R[x_1, \ldots, x_n]$ symbolically written as

$$\sum_{i=1}^{k} c_i x_1^{e_{1,i}} \cdots x_n^{e_{n,i}}$$

is represented by a sequence $[m_1, \ldots, m_k]$ of k monomials where each m_i is a pair

$$\langle c_i, [e_{1,i}, \ldots, e_{n,i}] \rangle$$

of a non-zero coefficient $c_i \in R$ and a sequence $[e_{1,i}, \ldots, e_{n,i}]$ of n exponents (natural numbers) representing the power product of the monomial. We sort the monomials $m_1, \ldots, m_k$ in reverse lexicographic order where monomial m_a occurs before monomial m_b, if for some variable index j we have $e_{a,j} > e_{b,j}$ and for every index $j' < j$ we have $e_{a,j'} = e_{b,j'}$. For instance, the polynomial $3xy^2 + 5x^2y + 7x + 11y + 13$ in $\mathbb{Z}[x, y]$ is represented by the sequence

$$[\langle 5, [2, 1] \rangle, \langle 3, [1, 2] \rangle, \langle 7, [1, 0] \rangle, \langle 11, [0, 1] \rangle, \langle 13, [0, 0] \rangle]$$

Every polynomial has thus a unique representation; please note that the zero polynomial is represented by the empty sequence.

In our program, we allow R to be any type that supports the usual ring operations, i.e., `Polynomial` is *generic* in its coefficient domain. For this proceed as follows:

1. Take the following abstract class `Ring`:

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

  // the string representation of this element
  virtual string str() = 0;

  // the constant zero of the type of this element
  virtual Ring* zero()  = 0;

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

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

2. Implement a concrete class `Integer`

```
class Integer: public Ring {
public:
  // integer with value n (default 0)
  Integer(long n=0);
};
```

This class overrides all the abstract (pure virtual) operations of class `Ring` by concrete definitions for integer arithmetic (where integers are represented by `long` values).

Note that the definition of the arithmetic and comparison functions the parameter `c` must be explicitly converted from type `Ring*` to type `Integer*`. Use the expression `dynamic_cast<Integer*>(c)` to receive a pointer to a `Integer` 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:
  // zero-polynomial in n variables with given names
  Polynomial(int n, string* vars);

  // add new term with given coefficient and exponents to this polynomial
  // and return this polynomial
  Polynomial& add(Ring* coeff, int* exps);

  // destructor
  virtual ~Polynomial();
};
```

which implements polynomials with generic coefficient types (i.e., coefficients that are represented by a concrete subclass of class `Ring`).

A `Polynomial` object shall be represented by

a) the number of variables,

b) a pointer to an array of the variable names,

c) a pointer to a heap-allocated array of monomials in these variables,

d) the length of this array,

e) the number of monomials in this array.

Each monomial holds a *pointer* to its coefficient (of type `Ring*`) and a pointer to the array of exponents. At any time, the array shall hold as many monomials as are indicated by the number value which is less than or equal the length value; these monomials hold coefficients different from 0, are unique with respect to their exponents and are sorted in reverse lexicographic order. It might be a good idea to introduce a class `Monomial` such that the monomial array in `Polynomial` holds elements of this type (respectively pointers to such elements, depending on your design choice).

When a new polynomial is created, we allocate an empty array of some default size. When a new monomial is added, we first check its coefficient; if it is zero, the monomial is ignored. Otherwise, we search in the array for the position where

a) either a monomial with the given exponent sequence already occurs; in this case, the given coefficient is added; if the resulting coefficient is zero, the monomial is discarded from the array and all subsequent monomials are shifted to fill the gap;

b) or, if there is no such exponent sequence, a new monomial is to be inserted; the subsequent monomials have to be shifted to make room for the new monomial.

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.

Please note the following:

- Class `Polynomial` does *not* use the class `Integer` described above, it is only based on the class `Ring` representing the coefficient domain.
- No memory leaks shall arise from the implementation. Thus the class needs a destructor that frees the memory allocated for the object.
- Check in the addition of polynomials whether the number of variables in both polynomials match; if not abort the program with an error message.
- For printing a polynomial remember that the interpretation of a polynomial with an empty monomial sequence is 0.

Avoid any code duplication but make extensive use of auxiliary functions (that shall become as far as possible private member functions of `Polynomial`). Write the declaration of `Polynomial` into a file `Polynomial.h` and the implementation of all non-trivial member functions of `Polynomial` and of into a file `Polynomial.cpp`.

Write a file `PolynomialMain.cpp` that uses `Polynomial` and tests its operations comprehensively. Test each operation with at least three test cases that also include special cases (such as adding a zero polynomial). Test the program also with the following piece of code:

```
// variable names and exponent vectors ("power products")
string vars[2] = { "x", "y" };
int e1[2] = {1,2}; int e2[2] = {2,1}; int e3[2] = {1,0};
int e4[2] = {0,1}; int e5[2] = {0,0}; int e6[2] = {2,2};

// construct polynomials p and q in two variables
Polynomial* p = new Polynomial(2, vars);
p->add(new Integer(3),e1).add(new Integer(5),e2).add(new Integer(7),e3)
  .add(new Integer(11),e4).add(new Integer(13),e5);
Polynomial* q = new Polynomial(2, vars);
q->add(new Integer(11),e4).add(new Integer(-3),e2).add(new Integer(2),e6)
  .add(new Integer(-2),e2);

// print p and q
cout << p->str() << endl << q->str() << endl;

// set p to p+2*q and print it
Polynomial* r = p->operator+(q)->operator+(q);
cout << r->str() << endl;
```