Object-Oriented Programming in C++ (SS 2021)
Exercise 2: April 29, 2021

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 2: Integers and Rationals

1. First, implement a class `Integer` whose objects represents integer numbers of arbitrary size; this class shall provide (at least) the following interface:

```
class Integer {
public:
  // integer number representing value <i>, default 0
  Integer(int i = 0);

  // integer number with sign <s> and <n> base-100 digits d[0],...
  Integer(bool s, int n, char *d);

  // destructor
  ~Integer();

  // overload output operator for this type
  friend ostream& operator<<(ostream& os, Integer& i);

  // negation, sum, difference, product of this number and <i>
  Integer operator-();
  Integer operator+(Integer& i);
  Integer operator-(Integer& i);
  Integer operator*(Integer& i);

  // equality and less-than-equality of this number and <i>
  bool operator==(Integer& i);
  bool operator<=(Integer& i);
};
```

An integer shall be represented by a triple $\langle s, n, d \rangle$ where s is a boolean value that is true if the integer is negative, n is a natural number, and d is a (pointer to a) heap-allocated array of length n whose elements represent bytes (values of type `char`): every element $d[i]$ holds the two decimal digits of the integer with positional values 10^{2i} and 10^{2i+1} (i.e., we represent the integers in the number system with base 100). For instance, the number -12345 is represented by s = true, $n = 3$, and $d = [45, 23, 1]$ (i.e., $d[0] = 45, d[1] = 23, d[2] = 1$). If $n = 0$, the representation denotes the number 0, if $n > 0$, then $a[n-1] > 0$, i.e., the representation does not store leading zeros (and is thus canonical).

If the second constructor is called with s = true and $n = 0$, the sign s = false is stored. Furthermore, this constructor does not use the given array d as its internal representation but creates a copy and makes sure that this copy does not hold leading zeros. The destructor frees this copy.

For implementing the arithmetic operations, first allocate a temporary array for the digits of the result and then fill this array with the appropriate digits; finally construct the actual result from this array using the second constructor and free the temporary array. The operations are to be implemented with the usual "school algorithms"; for example, for computing the product, multiply the first number with every digit of the second number and add the (appropriately shifted) intermediate results.

2. Second, implement a class `Rational` whose objects represent rational numbers (with numerators and denominators of arbitrary size); this class shall provide (at least) the following interface:

```
class Rational {
public:
  // rational with numerator <n> and denominator <d> (both may be negative)
  Rational(Integer &n, Integer &d);

  // destructor
  ~Rational();

  // overload output operator for this type
  friend ostream& operator<<(ostream& os, Rational& r);

  // arithmetic on this number and <r>
  Rational operator-();
  Rational operator+(Rational& r);
  Rational operator-(Rational& r);
  Rational operator*(Rational& r);
  Rational operator/(Rational& r);

  // equality and less-than-equality of this number and <r>
  bool operator==(Rational& r);
  bool operator<=(Rational& r);
};
```

A rational shall be represented by a pair $\langle n, d\rangle$ where numerator n and denominator $d > 0$ are integers of arbitrary size (values of type `Integer`); n and d may have common divisors, thus this representation is not canonical (please note that the comparison $a_1/b_1 = a_2/b_2$ can be implemented by the test $a_1 \cdot b_2 = a_2 \cdot b_1$; similarly a test for inequality is possible).

If the constructor is called with $d = 0$, the program aborts with an error message. If $d < 0$, then both the signs of n and d are inverted.

The classes thus support the following operations:

```
char d1[] = { 45, 23, 1 };
char d2[] = { 99, 66, 33, 0, 0 };
Integer i1(true, 3, d1); Integer i2(false, 5, d2);
cout << i1 << " " << i2 << endl;
Integer i3 = -i1; Integer i4 = i1+i2;
cout << i3 << " " << i4 << endl;
Integer i5 = i1-i2; Integer i6 = i1*i2;
cout << i5 << " " << i6 << endl;

Rational r1(i1,i2); Rational r2(i4,i5);
```

```
cout << r1 << " " << r2 << endl;
Rational r3 = -r1; Rational r4 = r1+r2; Rational r5 = r1-r2;
cout << r3 << " " << r4 << " " << r5 << endl;
Rational r6 = r1*r2; Rational r7 = r1/r2;
cout << r6 << " " << r7 << endl;
```

Test each of the two classes in a *comprehensive* way (several calls of each method) including also the calls shown above (print the results and show the program output in the deliverable).