Eidgenössische Technische Hochschule Zürich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Swiss Federal Institute of Technology Zurich

Institut für theoretische Informatik
Dr. B. Gärtner

17. Januar 2005

Informatik für Mathematiker und Physiker Serie 12 WS 04/05

URL: `http://www.ti.inf.ethz.ch/ew/courses/Info1_04/`

Aufgabe 1 (12 Punkte)

Entwerfen und implementieren Sie eine Klasse `Integer` zur Darstellung und Verarbeitung von ganzen Zahlen "beliebiger" (endlicher) Grösse. Die Klasse sollte folgende Funktionalität bieten.

- Konstruktion von `int` und `unsigned int`.
- Vergleichs– und Ordnungsrelationen: `==`, `!=`, `<`, `<=`, `>`, `>=`.
- Addition und Subtraktion.
- Ausgabe auf `std::ostream`.

Zur Abgabe gehört – wie immer – ein geeignetes Testprogramm.

Vorschläge:

- Repräsentieren Sie eine Zahl als ein Vorzeichen sowie einen Vektor von Ziffern zu einer beliebigen aber fixen Basis `base`.
- In der Bibliothek `<deque>` wird eine Klasse names `std::deque` definiert. Sie ist `std::vector` sehr ähnlich. Nur kann man in einer `std::deque` sowohl hinten (`push_back` und `pop_back`) als auch vorne (`push_front` und `pop_front`) einfügen und löschen.

Zusatzaufgabe 2 (8 Punkte)

Diese Punkte zählen nicht zur erreichbaren Gesamtpunktzahl, werden aber voll angerechnet.

Implementieren Sie für die Klasse aus Aufgabe 1 auch Multiplikation, Division und die Modulo-Operation.

Abgabe: Aufgaben 1+2: bis 30. Januar 2005, 16:00 Uhr, per email.
In der nächsten Woche wird es eine Schnellübung geben!

Programm: rational.h

```
// Programm: rational.h
// Brueche ganzer Zahlen.

#include <iostream>

namespace ifm {

  // Klasse zur Repraesentation von Bruechen ganzer Zahlen.
  class Rational {
  public:
    typedef int NT;

    Rational(const NT& n, const NT& d);
    // PRE: d != 0.
    // POST: Bruch initialisiert als n / d.

    Rational(const NT& n);
    // POST: Bruch initialisiert als n / 1.

    const NT& n() const;
    // POST: Rueckgabewert ist Zaehler des Bruchs.

    const NT& d() const;
    // POST: Rueckgabewert ist Nenner des Bruchs.

    Rational& operator+=(const Rational& x);
    // POST: x wurde zu *this addiert.

    Rational& operator-=(const Rational& x);
    // POST: x wurde von *this subtrahiert.

    Rational& operator*=(const Rational& x);
    // POST: *this wurde mit x multipliziert.

    Rational& operator/=(const Rational& x);
    // PRE: x.n() != 0
    // POST: *this wurde durch x dividiert.

  private:

    void normalize();
    // POST: Der Wert des Bruchs bleibt unveraendert, aber Zaehler und
    // Nenner sind teilerfremd, sowie der Nenner ist positiv.

    NT n_; // numerator (Zaehler)
    NT d_; // denominator (Nenner)

    // Invariante: d_ > 0 und (n_,d_) ist normalisiert.
  };

  std::ostream& operator<<(std::ostream& o, const Rational& x);
  // POST: x wurde auf o ausgegeben; Rueckgabewert ist o.

  // Vergleichsoperationen:
  // POST: Rueckgabewert true <=> x ? y
  bool operator==(const Rational& x, const Rational& y);
  bool operator!=(const Rational& x, const Rational& y);
  bool operator<(const Rational& x, const Rational& y);
  bool operator<=(const Rational& x, const Rational& y);
  bool operator>(const Rational& x, const Rational& y);
  bool operator>=(const Rational& x, const Rational& y);

  unsigned int gcd(unsigned int a, unsigned int b)
  // POST: Rueckgabe ist der groesste gemeinsame
  // Teiler von a und b, wobei ggt(x,0):=ggt(0,x):=x.
  {
    if (b == 0) return a;
    return gcd(b, a % b);
  } // gcd(a, b)

  //
  // Implementation der Klasse Rational
  //

  Rational::Rational(const NT& n, const NT& d) : n_(n), d_(d)
  { normalize(); }

  Rational::Rational(const NT& n) : n_(n), d_(1) {}

  const Rational::NT& Rational::n() const { return n_; }
  const Rational::NT& Rational::d() const { return d_; }

  void Rational::normalize()
  {
    if (d_ < 0) {
      d_ = -d_;
      n_ = -n_;
    }
    NT na = n_;
    if (n_ < 0) na = -na;
    NT g = ifm::gcd(na, d_);
    n_ /= g;
    d_ /= g;
  }

  Rational& Rational::operator+=(const Rational& x)
  {
    n_ = n_ * x.d_ + x.n_ * d_;
    d_ *= x.d_;
    normalize();
    return *this;
  }

  Rational& Rational::operator-=(const Rational& x)
  {
    n_ = n_ * x.d_ - x.n_ * d_;
    d_ *= x.d_;
    normalize();
    return *this;
  }

  Rational& Rational::operator*=(const Rational& x)
  {
    n_ *= x.n_;
    d_ *= x.d_;
    normalize();
    return *this;
  }

  Rational& Rational::operator/=(const Rational& x)
  {
    assert(x.n() != 0);
    n_ *= x.d_;
    d_ *= x.n_;
    normalize();
    return *this;
  }

  Rational operator+(const Rational& x, const Rational& y)
  // POST: Rueckgabewert ist x + y.
  {
    Rational z = x;
    z += y;
    return z;
  }

  Rational operator-(const Rational& x, const Rational& y)
  // POST: Rueckgabewert ist x - y.
  {
    Rational z = x;
    z -= y;
    return z;
  }

  Rational operator*(const Rational& x, const Rational& y)
  // POST: Rueckgabewert ist x * y.
  {
    Rational z = x;
    z *= y;
    return z;
  }

  Rational operator/(const Rational& x, const Rational& y)
  // POST: Rueckgabewert ist x / y.
  {
    Rational z = x;
    z /= y;
    return z;
  }

  //
  // Globale Funktionen
  //

  bool operator==(const Rational& x, const Rational& y)
  { return x.n() * y.d() == x.d() * y.n(); }

  bool operator!=(const Rational& x, const Rational& y)
  { return !(x == y); }

  bool operator<(const Rational& x, const Rational& y)
  { return x.n() * y.d() < x.d() * y.n(); }

  bool operator<=(const Rational& x, const Rational& y)
  { return !(y < x); }

  bool operator>(const Rational& x, const Rational& y)
  { return y < x; }

  bool operator>=(const Rational& x, const Rational& y)
  { return !(x < y); }

  std::ostream& operator<<(std::ostream& o, const Rational& x)
  { return o << x.n() << "/" << x.d(); }

} // namespace ifm
```