Solution to Exercise 107.

```
1  // Prog: rational.h
2  // Define a class for rational numbers
3  #include <iostream>
4  #include <cassert>
5
6  namespace math {
7
8    // gcd function
9    int gcd(int a, int b)
10   {
11     // PRE: (a,b) != (0,0)
12     // POST: return value is the greatest common divisor
13     //       of a and b, where gcd(n,0):=gcd(0,n):=n.
14     if (b == 0) return a;
15     return gcd(b, a % b);
16   }
17
18   // Rationals
19   class Rational {
20   public:
21     Rational();
22     // POST: *this is initialized with value 0
23
24     Rational(int n, int d);
25     // PRE: d != 0
26     // POST: *this is initialized with value n / d
27
28     Rational(int n);
29     // POST: *this is initialized with value n
30
31     int n();
32     // POST: return value is the numerator n
33
34     int d();
35     // POST: return value is the denominator d
36
37     Rational& operator+= (Rational s);
38     // POST: *this has been incremented by s, and
39     //       the return value is the new value
40
41     Rational& operator-= (Rational s);
42     // POST: *this has been decremented by s, and
43     //       the return value is the new value
44
45     Rational& operator*= (Rational s);
46     // POST: *this has been multiplied by s, and
47     //       the return value is the new value
48
49     Rational& operator/= (Rational s);
50     // POST: *this has been divided by s, and
51     //       the return value is the new value
52
53   private:
54     int n_;  // numerator
55     int d_;  // denominator; invariant: d_ != 0
56
57     void normalize();
58     // POST: gcd(_n,_d) is 1 or -1
59   };
60
61   Rational::Rational()
62     : n_(0), d_(1)
```

```
63    {}
64
65    Rational::Rational(int n, int d)
66      : n_(n), d_(d)
67    {
68      assert (d_ != 0);
69    }
70
71    Rational::Rational(int n)
72      : n_(n), d_(1)
73    {}
74
75    int Rational::n() { return n_; }
76    int Rational::d() { return d_; }
77
78    Rational& Rational::operator+= (Rational s)
79    {
80      n_ = n_ * s.d_ + d_ * s.n_;   // increment *this by s
81      d_ *= s.d_;
82      normalize();
83      return *this;                 // and return it
84    }
85
86    Rational& Rational::operator-= (Rational s)
87    {
88      n_ = n_ * s.d_ - d_ * s.n_;   // decrement *this by s
89      d_ *= s.d_;
90      normalize();
91      return *this;                 // and return it
92    }
93
94    Rational& Rational::operator*= (Rational s)
95    {
96      n_ *= s.n_;    // multiply *this by s
97      d_ *= s.d_;
98      normalize();
99      return *this;  // and return it
100   }
101
102   Rational& Rational::operator/= (Rational s)
103   {
104     n_ *= s.d_;    // multiply *this by 1/s
105     d_ *= s.n_;
106     normalize();
107     return *this;  // and return it
108   }
109
110   // normalization
111   void Rational::normalize()
112   {
113     int g = gcd(n_,d_);
114     n_ /= g;
115     d_ /= g;
116   }
117
118   // non-member arithemtic operators
119   Rational operator+(Rational r, Rational s)
120   {
121     Rational t = r;
122     return t += s;
123   }
124
125   Rational operator-(Rational r, Rational s)
126   {
127     Rational t = r;
```

```
128       return t -= s;
129     }
130
131     Rational operator*(Rational r, Rational s)
132     {
133       Rational t = r;
134       return t *= s;
135     }
136
137     Rational operator/(Rational r, Rational s)
138     {
139       Rational t = r;
140       return t /= s;
141     }
142
143     // non-member relational operators
144     bool operator==(Rational r, Rational s)
145     {
146       return r.n() * s.d() ==  s.n() * r.d();
147     }
148
149     bool operator!=(Rational r, Rational s)
150     {
151       return !(r == s);
152     }
153
154     bool operator<(Rational r, Rational s)
155     {
156       int a = r.n() * s.d();
157       int b = s.n() * r.d();
158       // watch out for the signs of denominators!
159       if (r.d() > 0)
160         if (s.d() > 0) return a < b; else return a > b;
161       else
162         if (s.d() > 0) return a > b; else return a < b;
163     }
164
165     bool operator>(Rational r, Rational s)
166     {
167       return s < r;
168     }
169
170     bool operator<=(Rational r, Rational s)
171     {
172       return r == s || r < s;
173     }
174
175     bool operator>=(Rational r, Rational s)
176     {
177       return r == s || s < r;
178     }
179
180     std::ostream& operator<<(std::ostream& o, Rational r)
181     {
182       return o << r.n() << "/" << r.d();
183     }
184   }
```

```
1   #include<iostream>
2   #include<cassert>
3   #include "rational_full.h"
4
5   int main () {
```

```
6    math::Rational r(3,4);
7    math::Rational s(-1,2);
8    math::Rational t = r * s; // -3/8
9    math::Rational u = r + s; // 1/4
10   math::Rational v = r - s; // 5/4
11   math::Rational w = r / s; // -3/2
12
13   // test arithemtic and equality operators
14   assert (t == math::Rational (-3,8));
15   assert (u == math::Rational (1,4));
16   assert (v == math::Rational (5,4));
17   assert (w == math::Rational (-3,2));
18
19   // test other relational operators
20   assert (s < r); assert (s != r);
21   assert (v > u); assert (v != u);
22   assert (u > w); assert (u != w);
23   assert (t > w); assert (t != w);
24
25   assert (t <= math::Rational (-3,8));
26   assert (u <= math::Rational (1,4));
27   assert (v >= math::Rational (5,4));
28   assert (w >= math::Rational (-3,2));
29
30   return 0;
31 }
```

Solution to Exercise 109.

```
1  #include <iostream>
2
3  // Class for representing time in hours : minutes : seconds
4  class Clock {
5  public: // error fix: public specifier was missing
6    Clock(unsigned int h, unsigned int m, unsigned int s);
7    // error fix: no reference types here!
8    // PRE: h < 24, m < 60, s < 60
9    // POST: *this is initialized with the time h:m:s
10
11   void tick();
12   // POST: time stored by *this has ben advanced by 1 second
13
14   void time(unsigned int& h, unsigned int& m,
15             unsigned int& s); // error fix: we need reference types!
16   // POST: h, m, s are filled from the time stored by *this
17 private:
18   unsigned int h_;
19   unsigned int m_;
20   unsigned int s_;
21 };
22
23 Clock::Clock(unsigned int h,
24              unsigned int m,
25              unsigned int s)
26     : h_(h), m_(m), s_(s)
27 {
28   assert (h < 24);
29   assert (m < 60);
30   assert (s < 60); // error fix: enforce preconditions
31 }
32
33 void Clock::tick()
34 {
```

```
35    h_ += (m_ += (s_ += 1) / 60) / 60;
36    h_ %= 24; m_ %= 60; s_ %= 60;
37  }
38
39  void Clock::time(unsigned int& h,
40                   unsigned int& m,
41                   unsigned int& s)
42  {
43    h = h_;
44    m = m_;
45    s = s_;
46  }
47
48  std::ostream& operator<< (std::ostream& o, Clock c)
49  {
50    unsigned int h;
51    unsigned int m;
52    unsigned int s;
53    c.time(h, m, s);
54    o << h << ":";
55    if (s < 10) o << "0";
56    o << m << ":";
57    if (s < 10) o << "0";
58    o << s;
59    return o;
60  }
61
62  int main() {
63    Clock c1 (23, 9, 8);
64    c1.tick(); // error fix: tick has to be called for an object
65
66    unsigned int h;
67    unsigned int m;
68    unsigned int s;
69    c1.time(h, m, s); // error fix: time has to be called for an object
70
71    std::cout << c1 << "\n";
72
73    return 0;
74  }
```

Solution to Exercise 110.

```
1   // Program: random_triangle.C
2   //
3   // A graphical random process:
4   //
5   //   * Start at an arbitrary vertex of a triangle t
6   //
7   //   * At each step, draw the current point p and choose
8   //     as a next point the midpoint of p and a (uniformly)
9   //     randomly selected vertex of t.
10
11  #include <iostream>
12  #include <IFM/window>
13
14  namespace math {
15    class Random {
16    public:
17      Random(unsigned int a, unsigned int b,
18             unsigned int m, unsigned int x0);
19      // POST: *this initialized according to
20      //       the formula x_i = ( ax_{i-1} + b) mod m
```

```
21
22      double operator()();
23      // POST: return value is the next pseudorandom number
24      //       in the sequence
25
26    private:
27      unsigned int a_;
28      unsigned int b_;
29      unsigned int m_;
30      unsigned int xi_;
31    };
32
33    Random::Random(unsigned int a, unsigned int b,
34                   unsigned int m, unsigned int x0)
35      : a_(a), b_(b), m_(m), xi_(x0)
36    {}
37
38    double Random::operator()()
39    {
40      // compute xi
41      xi_ = (a_ * xi_ + b_) % m_;
42      // normalize to [0,1)
43      return double(xi_) / m_;
44    }
45  } // end namespace math
46
47  int main()
48  {
49    std::cout << "Seed for random number generator =? ";
50    unsigned int seed;
51    std::cin >> seed;
52    // use the ANSIC generator
53    math::Random rnd(1103515245u, 12345u, 2147483648u, seed);
54
55    std::cout << "Number of steps =? ";
56    unsigned int n;
57    std::cin >> n;
58
59    // the vertices of the triangle
60    ifm::Point v0(0, 0);
61    ifm::Point v1(512, 0);
62    ifm::Point v2(256, 512);
63
64    ifm::Point cur = v0; // current point
65    for (unsigned int i = 0; i < n; ++i) {
66      // output current point
67      ifm::wio << cur;
68      // choose (uniformly) a random vertex of the triangle
69      ifm::Point rndvert;
70      int r = int(rnd() * 3); // random integer in [0,2]
71      if (r == 0) rndvert = v0;
72      else if (r == 1) rndvert = v1;
73      else rndvert = v2;
74      // jump halfway to the chosen vertex
75      cur = ifm::Point((cur.x() + rndvert.x()) / 2,
76                       (cur.y() + rndvert.y()) / 2);
77    }
78    ifm::wio.wait_for_mouse_click();
79    return 0;
80  }
```

Solution to Exercise 111. If `unsigned int` computations overflow, they will still be correct modulo 2^{32} on a 32-bit system; this is guaranteed by the C++ standard. This means,

if u is the mathematically correct value, the computed value is of the form $u + k2^{32}$ for some k. But now it is easy to see that taking u modulo 2^{31} gives the same result as taking $u + k2^{32}$ modulo 2^{31}, simply because $2^{32} \bmod 2^{31} = 0$.