B.11 Classes

Solution to Exercise 142. Here are the header and implementation files.

```
XSym
0047
e4cf9ef25d847e4335714f6c635a692a
../libraries/librational/include/IFM/rational.h
```

```
XSym
0041
8837a753eb6f93e23f8da5b6a7ec7961
../libraries/librational/lib/rational.cpp
```

And here is a test program: it computes harmonic numbers backwards and forwards, showing that there is no difference in the two sums if rational numbers are used for the computations.

```
// Program: harmonic.cpp
// Compute the n-th harmonic number in two ways.

#include <iostream>
#include <IFM/rational.h>

int main()
{
  // Input
  std::cout << "Compute H_n for n =? ";
  unsigned int n;
  std::cin >> n;

  // Forward sum
  math::rational fs (0, 1);
  for (unsigned int i = 1; i <= n; ++i)
    fs += math::rational (1, i);

  // Backward sum
  math::rational bs (0, 1);
  for (unsigned int i = n; i >= 1; --i)
    bs += math::rational (1, i);

  // Output
  std::cout << "Forward sum  = " << fs << "\n"
            << "Backward sum = " << bs << "\n";
  return 0;
}
```

Solution to Exercise 144.

```
#include <iostream>
#include <cassert>

// Class for representing time in hours : minutes : seconds
class Clock {
public: // error fix: public specifier was missing
  Clock(const unsigned int h, const unsigned int m, const unsigned int s);
```

```
 8    // make it const-correct!
 9    // error fix: no reference types here!
10    // PRE: h < 24, m < 60, s < 60
11    // POST: *this is initialized with the time h:m:s
12
13    void tick();
14    // POST: time stored by *this has ben advanced by 1 second
15
16    void time(unsigned int& h, unsigned int& m,
17              unsigned int& s) const; // error fix: we need reference types!
18    // POST: h, m, s are filled from the time stored by *this
19  private:
20    unsigned int h_;
21    unsigned int m_;
22    unsigned int s_;
23  };
24
25  Clock::Clock(const unsigned int h,
26               const unsigned int m,
27               const unsigned int s)
28      : h_(h), m_(m), s_(s)
29  {
30    assert (h < 24);
31    assert (m < 60);
32    assert (s < 60); // error fix: enforce preconditions
33  }
34
35  void Clock::tick()
36  {
37    h_ += (m_ += (s_ += 1) / 60) / 60;
38    h_ %= 24; m_ %= 60; s_ %= 60;
39  }
40
41  void Clock::time(unsigned int& h,
42                   unsigned int& m,
43                   unsigned int& s) const
44  {
45    h = h_;
46    m = m_;
47    s = s_;
48  }
49
50  std::ostream& operator<< (std::ostream& o, const Clock c)
51  {
52    unsigned int h;
53    unsigned int m;
54    unsigned int s;
55    c.time(h, m, s);
56    o << h << ":";
57    if (s < 10) o << "0";
58    o << m << ":";
59    if (s < 10) o << "0";
60    o << s;
61    return o;
62  }
63
64  int main() {
65    Clock c1 (23, 9, 8);
66    c1.tick(); // error fix: tick has to be called for an object
67
68    unsigned int h;
69    unsigned int m;
70    unsigned int s;
71    c1.time(h, m, s); // error fix: time has to be called for an object
72
```

```
73    std::cout << c1 << "\n";
74
75    return 0;
76  }
```

Solution to Exercise 145.

```
1   // Program: random_triangle.cpp
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  class Random {
15  public:
16    Random(const unsigned int a, const unsigned int b,
17           const unsigned int m, const unsigned int x0);
18    // POST: *this initialized according to
19    //       the formula x_i = ( ax_{i-1} + b) mod m
20
21    double operator()();
22    // POST: return value is the next pseudorandom number
23    //       in the sequence
24
25  private:
26    const unsigned int a_;
27    const unsigned int b_;
28    const unsigned int m_;
29    unsigned int xi_;
30  };
31
32  Random::Random(const unsigned int a, const unsigned int b,
33                 const unsigned int m, const unsigned int x0)
34    : a_(a), b_(b), m_(m), xi_(x0)
35  {}
36
37  double Random::operator()()
38  {
39    // compute xi
40    xi_ = (a_ * xi_ + b_) % m_;
41    // normalize to [0,1)
42    return double(xi_) / m_;
43  }
44
45  int main()
46  {
47    std::cout << "Seed for random number generator =? ";
48    unsigned int seed;
49    std::cin >> seed;
50    // use the ANSIC generator
51    Random rnd(1103515245u, 12345u, 2147483648u, seed);
52
53    std::cout << "Number of steps =? ";
54    unsigned int n;
55    std::cin >> n;
56
```

```
57    // the vertices of the triangle
58    const ifm::Point v0(0, 0);
59    const ifm::Point v1(512, 0);
60    const ifm::Point v2(256, 512);
61
62    ifm::Point cur = v0; // current point
63    for (unsigned int i = 0; i < n; ++i) {
64      // output current point
65      ifm::wio << cur;
66      // choose (uniformly) a random vertex of the triangle
67      ifm::Point rndvert;
68      int r = int(rnd() * 3); // random integer in [0,2]
69      if (r == 0) rndvert = v0;
70      else if (r == 1) rndvert = v1;
71      else rndvert = v2;
72      // jump halfway to the chosen vertex
73      cur = ifm::Point((cur.x() + rndvert.x()) / 2,
74                       (cur.y() + rndvert.y()) / 2);
75    }
76    ifm::wio.wait_for_mouse_click();
77    return 0;
78  }
```

Solution to Exercise 146. If `unsigned int` computations overflow, they will still be correct modulo 2^{32} on a 32-bit system (or any other system with at least this many bits); this is guaranteed by the C++ standard according to which `unsigned int` behaves like the ring Z_{32} under additions and multiplications. This means, if u is the mathematically correct value, the computed value is of the form $u + k2^{32}$, where k is the unique integer such that $u + k2^{32} \in \{0, \ldots, 31\}$.

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 $2^{32} \bmod 2^{31} = 0$.

Solution to Exercise 147. The opponent plays with a fair dice, i.e. he/she plays every number with an equal probability of $\frac{1}{6}$. The goal is to beat the fair dice. Let us for example consider the strategy of always choosing 1. Then our *average* gain (playing against the fair dice) can be computed from a table that lists what we get, depending on the number chosen by the opponent.

opponent's number	1	2	3	4	5	6
our gain (CHF) in choosing 1	0	2	-1	-1	-1	-1

Since the opponent plays each number with probability $1/6$, our average gain in choosing number 1 is

$$\frac{1}{6}(0 + 2 - 1 - 1 - 1 - 1) = \frac{-2}{6} = -\frac{1}{3}.$$

Not good.

But we can do the same for all other numbers and then end up with the following table.

our number	1	2	3	4	5	6
our average gain (CHF)	$-\frac{1}{3}$	$-\frac{1}{2}$	$-\frac{1}{6}$	$\frac{1}{6}$	$\frac{1}{2}$	$\frac{1}{3}$

Always choosing number 5 is therefore the best against a fair dice. This way, after n rounds of the game, we will have earned CHF $\frac{n}{2}$ on average.

In reality, our opponent would not be so stupid to stick with the fair dice if we always choose 5; he could switch to always choosing 4, making us lose in every round. But that's ok: the goal was to beat the (stupid) fair dice, and that's what we achieve by choosing 5.

Solution to Exercise 148. Here is a strategy that cannot lose in the long run, no matter what the opponent does. Choose the numbers according to the probability distribution

$$(p_1, p_2, p_3, p_4, p_5, p_6) = \left(0, \frac{1}{16}, \frac{5}{16}, \frac{1}{4}, \frac{5}{16}, \frac{1}{16}\right). \tag{B.18}$$

Now let's see what happens when the *opponent* plays some fixed number i. If we can show that our average gain is nonnegative, regardless of the opponent's number, then we have shown that we cannot lose in the long run. If the opponent plays 1, for example, we arrive at the following table.

our number	1	2	3	4	5	6
our gain (CHF) when opponent chooses 1	0	-2	1	1	1	1

Since we play as in (B.18), our average gain against the opponent choosing 1 is

$$0 \cdot 1 - \frac{1}{16} \cdot 2 + \frac{5}{16} \cdot 1 + \frac{1}{4} \cdot 1 + \frac{5}{16} \cdot 1 + \frac{1}{16} \cdot 1 = \frac{13}{16}.$$

Good.

We can do the same thing for all other numbers played by the opponent and then end up with the following (surprising) table.

opponent's number	1	2	3	4	5	6
our average gain (CHF)	$\frac{13}{16}$	0	0	0	0	0

Perfect! No matter which number the opponent chooses, we always earn at least CHF 0 on average. This means that we have found an unbeatable loaded dice.

Solution to Exercise 149. Let us start with the test program `xbm_merge.cpp`. It includes `transparency1.xbm` and `transparency2.xbm` and writes an `XBM` file (with header `overlay`) to standard output. **Note:** You have to recompile the program everytime you want to test with a different `XBM` input file!

```
#include<iostream>
#include<cassert>
#include "transparency1.xbm"
#include "transparency2.xbm"
#define width transparency1_width
#define height transparency1_height

int main()
{

  // array for mapping from {0,...,15} to {'0',...,'f'} for output
  char hex[] = {'0', '1', '2', '3', '4', '5', '6', '7',
                '8', '9', 'a', 'b', 'c', 'd', 'e', 'f'};

  // output header for merged image
  std::cout << "#define overlay_width "    << width << "\n";
  std::cout << "#define overlay_height "   << height << "\n";
  std::cout << "static unsigned char bits[] = {\n";

  bool comma = false;
  for (int i=0; i<sizeof(transparency1_bits); ++i) {
    unsigned char outbyte = 0;
    unsigned char transparency1 = transparency1_bits[i];
    unsigned char transparency2 = transparency2_bits[i];
    unsigned char fac=1;
    for (int j=0; j<8; ++j) {
      if ((transparency1 % 2) || (transparency2 % 2))
        outbyte += fac;
      transparency1/=2;
      transparency2/=2;
      fac *= 2;
    }
    if (comma) std::cout << ","; comma = true;
    std::cout << "0x" << hex[outbyte / 16] << hex[outbyte % 16];
  }
  std::cout << "};\n";
  return 0;
}
```

Now the actual encrypting program `visual_crypto.cpp`. It includes `original.xbm` and writes an XBM file with header `transparency1` to standard output, if called with `visual_crypto 1`. The call `visual_crypto 2` entails header `transparency2`. As with `xbm_merge.cpp`. you have to recompile the program everytime you want to test with a different XBM input file.

```
// Prog: visual_crypto.cpp
// includes an xbm image file and outputs two "random" xbm-files whose
// overlay corresponds to the original image; every pixel is replaced
// by four pixels in each image

#include<iostream>
#include<cassert>
#include<string>
#include<IFM/random.h>
#include "original.xbm"
#define inwidth original_width
#define inheight original_height
#define inbits original_bits

// POST: returns ", ", unless called for the first time where we get ""
std::string comma() {
```

```
17    static bool first_time = true;
18    if (!first_time)
19      return ", ";
20    else {
21      first_time = false;
22      return "";
23    }
24  }
25
26  // POST: return value is the hex-code of i for XBM output.
27  //       For example: 30 = 1*16 + 14 -> 0x1e
28  std::string hex (const unsigned char i) {
29    static std::string h[] = {"0", "1", "2", "3", "4", "5", "6", "7",
30                              "8", "9", "a", "b", "c", "d", "e", "f"};
31    return "0x" + h[i / 16] + h[i % 16];
32  }
33
34  // POST: computes the two-pixel replacement (01 = 1 or 10 = 2)
35  // of pixel p, for row 0 in image i, under (random) bit b, according
36  // to the following table. The replacement for row 1 is the reverse
37  //   bit              0 | 1           0 | 1
38  // ---------------------------------------------------
39  //   image 0:  0 -> 10 | 01;   1 -> 10 | 01   <- row 0
40  //                   01 | 10;        01 | 10   <- row 1
41  //
42  //   image 1:  0 -> 10 | 01;   1 -> 01 | 10   <- row 0
43  //                   01 | 10;        10 | 01   <- row 1
44  // ---------------------------------------------------
45  // => overlay: 0 -> 10 | 01;   1 -> 11 | 11   <- row 0
46  //                   01 | 10;        11 | 11   <- row 1
47  //                   (gray)          (black)
48  unsigned char split_pixel (const bool p, const bool i, const bool b)
49  {
50    if (p && i) return b ? 2 : 1; else return b ? 1 : 2;
51  }
52
53  // POST: computes the four replacement bytes for byte, for rows 0 and 1
54  //       in image i, using some fixed random number generator.
55  //       The replacement happens pixel by pixel, according to split_pixel.
56  //       The replacement bytes are written to outbyte01, outbyte02 for
57  //       row 0 and outbyte11, outbyte12 for row 1.
58  void split_byte (unsigned char byte, bool i,
59                   unsigned char& outbyte01, // for the first four pixels, row 0
60                   unsigned char& outbyte02, // for the last four pixels, row 0
61                   unsigned char& outbyte11, // for the first four pixels, row 1
62                   unsigned char& outbyte12) // for the last four pixels, row 1
63  {
64    // the random number generator; it is important that the sequence of
65    // random numbers is the same for every instance, since image 1 and 2
66    // need to be correlated
67    static ifm::random ansic (1103515245u, 12345u, 2147483648u, 12345u);
68
69    // go through the pixels and build the outbytes
70    unsigned char outbyte[4];
71    for (int k=0; k<2; ++k) {
72      outbyte[k] = outbyte[k+2] = 0; // 0/2: first bytes, 1/3: second bytes
73      unsigned char power4 = 1;      // for prepending 01 or 10 to outbyte
74      for (int j=0; j<4; ++j) {
75        const bool p = byte % 2;           // last bit = next pixel
76        // prepend replacement bits to outbytes
77        const unsigned char split =  split_pixel (p, i, ansic()<0.5);
78        outbyte[k]   += power4 * split;     // row 0
79        outbyte[k+2] += power4 * (3-split); // row 1: 1 <-> 2
80        // shift power for next round
81        power4 *= 4;
```

```
82        // chop off processed bit
83        byte /= 2;
84      }
85    }
86    outbyte01 = outbyte[0];
87    outbyte02 = outbyte[1];
88    outbyte11 = outbyte[2];
89    outbyte12 = outbyte[3];
90  }
91
92  int main(int argc, char* argv[])
93  {
94    if (argc < 2) {
95      // no command line arguments (except program name)
96      std::cout << "Usage: visual_crypto n (where n is 1 or 2)\n";
97      return 1;
98    }
99
100   // read number of image to be generated (second command line argument)
101   char* const s = argv[1];
102   bool i;    // image number
103   if      (*s == '1') i = 0;
104   else if (*s == '2') i = 1;
105   else {
106     std::cerr << "Illegal argument (should be 0 or 1)\n";
107     return 1;
108   }
109
110   // output header for encrypted image, containing number of image
111   std::cout << "#define transparency"
112             << *s << "_width "  << 2*inwidth << "\n";
113   std::cout << "#define transparency"
114             << *s << "_height " << 2*inheight << "\n";
115   std::cout << "static unsigned char transparency"
116             << *s << "_bits[] = {\n";
117
118   const int bytes_per_inrow =  (inwidth+7)/8;
119   const int bytes_per_outrow =
120     2 * bytes_per_inrow - (inwidth % 8 > 0 && inwidth % 8 <= 4);
121   // go through the input image row by row;
122   for (int row = 0; row < inheight; ++row) {
123     // compute the two replacement rows
124     unsigned char outbytes0[2*bytes_per_inrow]; // replacement bytes, row 0
125     unsigned char outbytes1[2*bytes_per_inrow]; // replacement bytes, row 1
126     for (int b = 0; b < bytes_per_inrow; ++b)
127       // split b-th input byte
128       split_byte (inbits [row * bytes_per_inrow + b], i,
129                   outbytes0[2*b], outbytes0[2*b+1],
130                   outbytes1[2*b], outbytes1[2*b+1]);
131     // write out the replacement rows
132     for (int b = 0; b < bytes_per_outrow; ++b)
133       std::cout << comma() << hex(outbytes0[b]); // row 0
134     for (int b = 0; b < bytes_per_outrow; ++b)
135       std::cout << comma() << hex(outbytes1[b]); // row 1
136   }
137   std::cout << "};\n";
138
139   return 0;
140
141 }
```