Solution to Exercise 93.

There are several possible representations for three-valued logic. A convenient one is based on the following observation: if we interpret *false* as 0, *unknown* as 1 and *true* as 2, then AND corresponds to the minimum of the two numbers, while OR corresponds to the maximum. The following program defines the type `Tribool` and the two operators, and it uses them to reproduce the truth tables.

```
1  // Prog: Tribool.C
2  // imp,ements three-valued logic
3
4  #include<iostream>
5  #include<cassert>
6
7  struct Tribool {
8    // INV : value in {0, 1, 2}
9    // 0 = false, 1 = unknown, 2 = true
10   unsigned int value;
11 };
12
13 // PRE: val in {0, 1, 2}
14 // POST: return value is a Tribool with the corresponding value
15 Tribool tribool (unsigned int val)
16 {
17   assert (val <= 2);
18   Tribool result;
19   result.value = val;
20   return result;
21 }
22
23 // POST: returns x AND y
24 Tribool operator&& (Tribool x, Tribool y)
25 {
26   Tribool result;
27   if (x.value < y.value)
28     result.value = x.value;
29   else
30     result.value = y.value;
31   return result;
32 }
33
34 // POST: returns x OR y
35 Tribool operator|| (Tribool x, Tribool y)
36 {
37   Tribool result;
38   if (x.value > y.value)
39     result.value = x.value;
40   else
41     result.value = y.value;
42   return result;
43 }
44
45 // POST: Tribool value is written to std::cout
46 void print (Tribool x)
47 {
48   if      (x.value == 0)  std::cout << "false   ";
49   else if (x.value == 1)  std::cout << "unknown ";
50   else                    std::cout << "true    ";
51 }
52
53 int main()
54 {
```

```
55    // print 3 x 3 truth table for AND
56    for (int x_val = 0; x_val < 3; ++x_val) {
57      Tribool x = tribool (x_val);
58      for (int y_val = 0; y_val < 3; ++y_val) {
59        Tribool y = tribool (y_val);
60        print (x && y);
61      }
62      std::cout << "\n";
63    }
64    std::cout << "\n";
65
66    // print 3 x 3 truth table for OR
67    for (int x_val = 0; x_val < 3; ++x_val) {
68      Tribool x = tribool (x_val);
69      for (int y_val = 0; y_val < 3; ++y_val) {
70        Tribool y = tribool (y_val);
71        print (x || y);
72      }
73      std::cout << "\n";
74    }
75    std::cout << "\n";
76
77
78    return 0;
79  }
```

Solution to Exercise 94.

The most natural representation is by an `unsigned int` value in the range $\{0, \ldots, 6\}$. Addition then simply adds the values and takes the result modulo 7. Subtraction could in principle be realized by subtratcing the values and taking the result modulo 7, but the problem is that the intermediate subtraction result could be negative (and therefore not representable as `unsigned int` value). To overcome this problem, we simply add 7 to the first value before we subtract the second one; this is guaranteed to yield a positive value, and modulo 7, it makes no difference. The following program defines the type `Z_7` and the two operators, and it uses them to reproduce the addition and subtraction table.

```
1   #include<iostream>
2
3   struct Z_7 {
4     // INV : value in {0, 1, 2, 3, 4, 5, 6}
5     unsigned int value;
6   };
7
8   // POST: return value is the sum of a and b
9   Z_7 operator+ (Z_7 a, Z_7 b)
10  {
11    Z_7 result;
12    result.value = (a.value + b.value) % 7;
13    return result;
14  }
15
16  // POST: return value is the difference of a and b
17  Z_7 operator- (Z_7 a, Z_7 b)
18  {
19    Z_7 result;
20    result.value = (7 + a.value - b.value) % 7;
21    return result;
```

```
22  }
23
24  int main ()
25  {
26    Z_7 a;
27    Z_7 b;
28
29    // print table for addition
30    for (unsigned int a_val = 0; a_val < 7; ++a_val) {
31      for (unsigned int b_val = 0; b_val < 7; ++b_val) {
32        a.value = a_val;
33        b.value = b_val;
34        std::cout << (a + b).value << " ";
35      }
36      std::cout << "\n";
37    }
38    std::cout << "\n";
39
40    // print table for subtraction
41    for (unsigned int a_val = 0; a_val < 7; ++a_val) {
42      for (unsigned int b_val = 0; b_val < 7; ++b_val) {
43        a.value = a_val;
44        b.value = b_val;
45        std::cout << (a - b).value << " ";
46      }
47      std::cout << "\n";
48    }
49    std::cout << "\n";
50
51    return 0;
52  }
```

Solution to Exercise 95. The following program is at the same time the solution to the next Exercise 96.

```
1   // Program: rational.C
2   // Define and use operations on rational numbers.
3
4   #include <iostream>
5
6   // the new type Rational
7   struct Rational {
8     int n;
9     int d; // INV: d != 0
10  };
11
12  // POST: return value is the sum of a and b
13  Rational operator+ (Rational a, Rational b)
14  {
15    Rational result;
16    result.n = a.n * b.d + a.d * b.n;
17    result.d = a.d * b.d;
18    return result;
19  }
20
21  // POST: return value is the difference of a and b
22  Rational operator- (Rational a, Rational b)
23  {
24    Rational result;
25    result.n = a.n * b.d - a.d * b.n;
26    result.d = a.d * b.d;
27    return result;
```

```
28  }
29
30  // POST: return value is the product of a and b
31  Rational operator* (Rational a, Rational b)
32  {
33    Rational result;
34    result.n = a.n * b.n;
35    result.d = a.d * b.d;
36    return result;
37  }
38
39  // POST: return value is the quotient of a and b
40  // PRE:  b != 0
41  Rational operator/ (Rational a, Rational b)
42  {
43    Rational result;
44    result.n = a.n * b.d;
45    result.d = a.d * b.n;
46    return result;
47  }
48
49  // POST: return value is true if and only if a == b
50  bool operator== (Rational a, Rational b)
51  {
52    return a.n * b.d == a.d * b.n;
53  }
54
55  // POST: return value is true if and only if a != b
56  bool operator!= (Rational a, Rational b)
57  {
58    return !(a == b);
59  }
60
61  // POST: return value is true if and only if a < b
62  bool operator< (Rational a, Rational b)
63  {
64    // here we have to watch out for signs
65    if (a.d > 0 && b.d > 0 || a.d < 0 && b.d < 0)
66      // no sign reversal in multplying by a.d and b.d
67      return a.n * b.d < a.d * b.n;
68    else
69      // sign reversal
70      return a.n * b.d > a.d * b.n;
71  }
72
73  // POST: return value is true if and only if a <= b
74  bool operator<= (Rational a, Rational b)
75  {
76    return a < b || a == b;
77  }
78
79  // POST: return value is true if and only if a > b
80  bool operator> (Rational a, Rational b)
81  {
82    return b < a;
83  }
84
85  // POST: return value is true if and only if a >= b
86  bool operator>= (Rational a, Rational b)
87  {
88    return a > b || a == b;
89  }
90
91
92  // POST: a has been written to o
```

```
93  std::ostream& operator<< (std::ostream& o, Rational a)
94  {
95    return o << a.n << "/" << a.d;
96  }
97
98  // POST: a has been read from i
99  // PRE:  i starts with a rational number of the form "n/d"
100 std::istream& operator>> (std::istream& i, Rational& a)
101 {
102   char c; // separating character, e.g. '/'
103   return i >> a.n >> c >> a.d;
104 }
105
106 int main ()
107 {
108   // input
109   std::cout << "Rational number r:\n";
110   Rational r;
111   std::cin >> r;
112
113   std::cout << "Rational number s:\n";
114   Rational s;
115   std::cin >> s;
116
117   // test the operations
118   std::cout << "Sum is         " << r + s    << ".\n";
119   std::cout << "Difference is  " << r - s    << ".\n";
120   std::cout << "Product is     " << r * s    << ".\n";
121   std::cout << "Quotient is    " << r / s    << ".\n";
122   std::cout << "r == s?        " << (r == s) << ".\n";
123   std::cout << "r != s?        " << (r != s) << ".\n";
124   std::cout << "r < s?         " << (r < s)  << ".\n";
125   std::cout << "r <= s?        " << (r <= s) << ".\n";
126   std::cout << "r > s?         " << (r > s)  << ".\n";
127   std::cout << "r >= s?        " << (r >= s) << ".\n";
128
129   return 0;
130 }
```

Solution to Exercise 96. See solution to Exercise 95.

Solution to Exercise 97. It turns out that only `operator+` requires real work; the following program evaluates it according to the case distinction in the small table that is given.

```
1  #include<iostream>
2
3  struct extended_int {
4    unsigned int u;  // the absolute value
5    bool         n;  // the sign (true means negative)
6  };
7
8  // POST: return value is the sum of a and b
9  //              ||  a > b      ||  a <= b
10 // ====================================
11 // (a,+)+(b,+) ||          (a+b,+)
12 // (a,+)+(b,-) || (a-b,+)     ||  (b-a,-)
13 // (a,-)+(b,+) || (a-b,-)     ||  (b-a,+)
14 // (a,-)+(b,-) ||          (a+b,-)
15 extended_int operator+ (extended_int a, extended_int b)
16 {
17   extended_int result;
```

```
18    if (a.n == b.n) {
19      result.u = a.u + b.u;
20      result.n = a.n;
21    }
22    else
23      if (a.u > b.u) {
24        result.u = a.u - b.u;
25        result.n = a.n;
26      }
27      else {
28        result.u = b.u - a.u;
29        result.n = b.n;
30      }
31    return result;
32  }
33
34  // POST: return value is -a
35  extended_int operator- (extended_int a)
36  {
37    a.n = !a.n;
38    return a;
39  }
40
41  // POST: return value is the difference of a and b
42  extended_int operator- (extended_int a, extended_int b)
43  {
44    return a + (-b);
45  }
46
47  // POST: return value is the product of a and b
48  extended_int operator* (extended_int a, extended_int b)
49  {
50    extended_int result;
51    result.u = a.u * b.u;
52    result.n = (a.n || b.n) && !(a.n && b.n); // XOR
53    return result;
54  }
55
56  // POST: a has been written to o
57  std::ostream& operator<< (std::ostream& o, extended_int a)
58  {
59    if (a.n) o << "-";
60    return o << a.u;
61  }
62
63  // POST: a has been set to i
64  void set (extended_int& a, int i)
65  {
66    if (i < 0) {
67      a.u = -i; a.n = true;
68    } else {
69      a.u = i; a.n = false;
70    }
71  }
72
73  // now test it
74  int main() {
75    extended_int x;
76    extended_int y;
77    for (int i = -1; i < 2; ++i)
78      for (int j = -1; j < 2; ++j) {
79        // set x to i
80        set (x, i);
81        set (y, j);
82        std::cout << "x = " << x << ", y = " << y << "\n";
```

```
83        std::cout << " x + y = " << x + y << "\n";
84        std::cout << " x - y = " << x - y << "\n";
85        std::cout << " x * y = " << x * y << "\n";
86      }
87    return 0;
88  }
```

Solution to Exercise 98. Let us use the following shortcuts:

E exact match
P promotion
S standard conversion

Here is the table of match qualities for the two parameters.

	A	B	C
a)	(S,S)	**(S,E)**	(S,S)
b)	(S,P)	(E,S)	(S,S)
c)	(E,S)	(S,E)	(S,S)
d)	(S,S)	(S,S)	**(S,E)**
e)	**(S,P)**	(S,S)	(S,S)
f)	(P,E)	(S,S)	(E,S)

A best match is indicated in bold. This implies that a) resolves to B; b),c) are ambiguous; d) resolves to C; e) resolves to A; f) is ambiguous.

Solution to Exercise 100.

a) Since `i` is changed in the function body, S may only be one of `int` and `int&`. T can be any of the three types `int`, `int&` and `const int&`, since values of all three types can be initialized from the lvalue `++i` of type `int`.

b) If S is `int`, then T may only be `int`, since otherwise, the function returns a reference to a temporary object, namely the local copy of the formal parameter `i`. If S is `int&`, T can as before be any of the three types.

c) Here are the postconditions.

```
// POST: return value is i+1
int foo (int i);

// POST: i has been incremented by 1;
//       return value is the new value of i
int foo (int& i);

// POST: i has been incremented by 1 and
//       is returned as an lvalue
int& foo (int& i);

// POST: i has been incremented by 1 and
//       is returned as a non-modifiable lvalue
const int& foo (int& i);
```

Solution to Exercise 101.

```
#include <iostream>

// POST: the values of i and j are swapped
void swap (int& i, int& j)
{
  int h = i;
  i = j;
  j = h;
}

int main() {
  // input
  std::cout << "i =? ";
  int i; std::cin >> i;

  std::cout << "j =? ";
  int j; std::cin >> j;

  // function call
  swap(i, j);

  // output
  std::cout << "Values after swapping: i = " << i
            << ", j = " << j << ".\n";

  return 0;
}
```

Solution to Exercise 102. We implement the second version, the one that returns the normalization of `r`. This one has the advantage that it works for rvalues.

The modification of the function `gcd` is as easy as it can be: we only need to replace the type `unsigned int` by `int` in the parameter and return types. Why does this still work? Let us go back to the proof of Lemma 1. Going through it, we realize that we never used nonnegativity of either a or b, so the statement extends to all pairs of integers with $b \neq 0$. It remains to prove termination. For this, we show that $|a \bmod b| < |b|$, so we indeed make progress towards termination.

Recall that

$$a \bmod b = a - (a \operatorname{div} b)b,$$

and that this equation also holds in C++. Furthermore, division may round up or down (we don't know), but in either case, the rounding makes a mistake of strictly less than 1. This means that

$$\frac{a}{b} - (a \operatorname{div} b)$$

has aboslute value smaller than 1, and this implies (by multiplying with b) that

$$a - (a \operatorname{div} b)b = a \bmod b$$

has absolute value smaller than $|b|$.

```
#include <iostream>

struct Rational {
  int n;
  int d; // INV: d != 0
};

// POST: a has been written to o
std::ostream& operator<< (std::ostream& o, Rational a)
{
  return o << a.n << "/" << a.d;
}

// POST: a has been read from i
// PRE:  i starts with a rational number of the form "n/d"
std::istream& operator>> (std::istream& i, Rational& a)
{
  char c; // separating character, e.g. '/'
  return i >> a.n >> c >> a.d;
}

// POST: return value is the greatest common divisor of a and b
int gcd (int a, int b)
{
  if (b == 0) return a;
  return gcd(b, a % b);  // b != 0
}

// POST: return value is the normalization of r
Rational normalize (const Rational& r)
{
  int g = gcd (r.n, r.d);
  Rational result;
  result.n = r.n / g;
  result.d = r.d / g;
  if (result.d < 0) {
    result.n = -result.n;
    result.d = -result.d;
  }
  return result;
}

int main ()
{
  std::cout << "Rational number r =? ";
  Rational r;
  std::cin >> r;
  std::cout << "Normalization is " << normalize(r) << ".\n";

  return 0;
}
```

Solution to Exercise 103.

```
#include <iostream>

// POST: return value indicates whether the linear equation
//       a * x + b = 0 has a real solution x ; if true is
//       returned, the value s satisfies a * s + b = 0
bool solve (double a, double b, double& s)
{
  // we have a solution if a is nonzero (s = -b/a),
```

```
 9    // or if both a and b are zero (take s = 0 in this case)
10    if (a != 0.0) {
11      s = -b / a;
12      return true;
13    }
14    // now we have a == 0.0
15    if (b == 0.0) {
16      s = 0.0;
17      return true;
18    }
19    return false;
20  }
21
22  int main()
23  {
24    std::cout << "solve a * x + b = 0 for\n";
25    std::cout << "a =? ";
26    double a;
27    std::cin >> a;
28    std::cout << "b =? ";
29    double b;
30    std::cin >> b;
31
32    double s;
33    if (solve (a, b, s))
34      std::cout << "Solution is " << s << ".\n";
35    else
36      std::cout << "Sorry, there is no solution.\n";
37
38    return 0;
39  }
```

Solution to Exercise 104.

a) None, since both `a` and `b` change.

b) None, since both `n` and `s` change.

c) The variable `bound` could be declared of type `const int` (line 15).

d) The formal parameter `i` could be declared of type `const unsigned int` (line 9).

e) The variable `t` could be declared of type `const Rational` (line 34).

Solution to Exercise 105.

a) Problem: Initialization of non-const reference from const object. (The variable `i` is const-qualified and can, therefore, not be passed as a non-const reference argument to the function `foo`.)

b) ok. (The variable `j` is a const reference and may, therefore, be initialized from a temporary.)

c) Problem: Initialization of reference from temporary. (The return value of the function `foo` is of type `int` and, therefore, the corresponding object has temporary lifetime. Via the function `bar` that temporary is used to initialize the variable `j`.)

d) Problem: Initialization of non-const reference from const reference. (The function `bar` returns a const reference that cannot be passed as a non-const reference to the function `foo`.)

e) ok. (Remark: Does not violate the Single Modification Rule because there is a sequence point after all function arguments have been evaluated.)