Solution to Exercise 115.

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 116.

```
1  #include<iostream>
2
3  // POST: the values of i and j are swapped
4  void swap (int& i, int& j)
5  {
6    int h = i;
7    i = j;
8    j = h;
9  }
10
11 int main() {
12   // input
13   std::cout << "i =? ";
14   int i; std::cin >> i;
15
16   std::cout << "j =? ";
17   int j; std::cin >> j;
18
19   // function call
20   swap(i, j);
21
22   // output
23   std::cout << "Values after swapping: i = " << i
```

```
24              << ", j = " << j << ".\n";
25
26    return 0;
27  }
```

Solution to Exercise 117. 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 $|b|$.

```
1   #include <iostream>
2
3   struct Rational {
4     int n;
5     int d; // INV: d != 0
6   };
7
8   // POST: a has been written to o
9   std::ostream& operator<< (std::ostream& o, Rational a)
10  {
11    return o << a.n << "/" << a.d;
12  }
13
14  // POST: a has been read from i
15  // PRE:  i starts with a rational number of the form "n/d"
16  std::istream& operator>> (std::istream& i, Rational& a)
17  {
18    char c; // separating character, e.g. '/'
19    return i >> a.n >> c >> a.d;
20  }
21
22  // POST: return value is the greatest common divisor of a and b
23  int gcd (int a, int b)
24  {
```

```
25    if (b == 0) return a;
26    return gcd(b, a % b);  // b != 0
27  }
28
29  // POST: return value is the normalization of r
30  Rational normalize (const Rational& r)
31  {
32    int g = gcd (r.n, r.d);
33    Rational result;
34    result.n = r.n / g;
35    result.d = r.d / g;
36    if (result.d < 0) {
37      result.n = -result.n;
38      result.d = -result.d;
39    }
40    return result;
41  }
42
43  int main ()
44  {
45    std::cout << "Rational number r =? ";
46    Rational r;
47    std::cin >> r;
48    std::cout << "Normalization is " << normalize(r) << ".\n";
49
50    return 0;
51  }
```

Solution to Exercise 118.

```
 1  #include <iostream>
 2
 3  // POST: return value indicates whether the linear equation
 4  //       a * x + b = 0 has a real solution x ; if true is
 5  //       returned, the value s satisfies a * s + b = 0
 6  bool solve (double a, double b, double& s)
 7  {
 8    // 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 119.

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 120.

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.)

Solution to Exercise 121.

```
1  // Prog: solve_quadratic_equation.C
2  // computes both (possibly complex) solutions to a quadratic equation
3  #include <iostream>
4  #include <complex>
5
6  // POST: return value is the number of distinct complex solutions
7  //       of the quadratic equation ax^2 + bx + c = 0. If there
8  //       are infinitely many solutions (a=b=c=0), the return
```

```
9  //       value is -1. Otherwise, the return value is a number n
10 //       from {0,1,2}, and the solutions are written to s1,..,sn
11 int solve_quadratic_equation (std::complex<double> a,
12                               std::complex<double> b,
13                               std::complex<double> c,
14                               std::complex<double>& s1,
15                               std::complex<double>& s2)
16 {
17   if (a == 0.0)
18     // linear case: bx + c = 0
19     if (b == 0.0)
20       // trivial case: c = 0
21       if (c == 0.0)
22         return -1; // => infinitely many solutions
23       else
24         return 0;  // => no solution
25     else {
26       // bx + c = 0, b != 0   => one solution
27       s1 = -c/b;
28       return 1;
29     }
30   else {
31     // ax^2 + bx + c = 0, a != 0 => two solutions
32     std::complex<double> d = std::sqrt(b*b-4.0*a*c);
33     s1 = (-b + d) / (2.0*a);
34     s2 = (-b - d) / (2.0*a);
35     return 2;
36   }
37 }
38
39
40 int main()
41 {
42   // input
43   std::cout << "Solve quadratic equation ax^2 + bx + c = 0 for\n";
44   std::cout << "a =? ";
45   double a;
46   std::cin >> a;
47   std::cout << "b =? ";
48   double b;
49   std::cin >> b;
50   std::cout << "c =? ";
51   double c;
52   std::cin >> c;
53
54   // computation
55   std::complex<double> s1;
56   std::complex<double> s2;
57
58   int n = solve_quadratic_equation (a, b, c, s1, s2);
59
60   // output
61   std::cout << "Number of solutions: " << n << "\n";
62   std::cout << "Solutions:\n";
63   if (n > 0) std::cout << s1 << "\n";
64   if (n > 1) std::cout << s2 << "\n";
65
66   return 0;
67
68 }
```