B.10 Reference Types

Solution to Exercise 135.

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 136.

```
#include<iostream>

// POST: the values of i and j are swapped
void swap (int& i, int& j)
{
  const 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 137. 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 absolute 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, const 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 (const int a, const 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)
{
  const 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 138.

```
#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 (const double a, const double b, double& s)
{
  // we have a solution if a is nonzero (s = -b/a),
  // or if both a and b are zero (take s = 0 in this case)
  if (a != 0.0) {
    s = -b / a;
    return true;
  }
  // now we have a == 0.0
  if (b == 0.0) {
    s = 0.0;
    return true;
  }
  return false;
}

int main()
{
  std::cout << "solve a * x + b = 0 for\n";
  std::cout << "a =? ";
  double a;
  std::cin >> a;
  std::cout << "b =? ";
  double b;
  std::cin >> b;
```

```

  double s;
  if (solve (a, b, s))
    std::cout << "Solution is " << s << ".\n";
  else
    std::cout << "Sorry, there is no solution.\n";

  return 0;
}
```

Solution to Exercise 139.

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 140.

```
// Prog: solve_quadratic_equation.cpp
// computes both (possibly complex) solutions to a quadratic equation
#include<iostream>
#include<complex>

// POST: return value is the number of distinct complex solutions
//       of the quadratic equation ax^2 + bx + c = 0. If there
//       are infinitely many solutions (a=b=c=0), the return
//       value is -1. Otherwise, the return value is a number n
//       from {0,1,2}, and the solutions are written to s1,..,sn
int solve_quadratic_equation (const std::complex<double> a,
                              const std::complex<double> b,
                              const std::complex<double> c,
                              std::complex<double>& s1,
                              std::complex<double>& s2)
{
  if (a == 0.0)
    // linear case: bx + c = 0
    if (b == 0.0)
      // trivial case: c = 0
      if (c == 0.0)
        return -1; // => infinitely many solutions
```

```
      else
        return 0;  // => no solution
    else {
      // bx + c = 0, b != 0   => one solution
      s1 = -c/b;
      return 1;
    }
  else {
    // ax^2 + bx + c = 0, a != 0 => two solutions
    const std::complex<double> d = std::sqrt(b*b-4.0*a*c);
    s1 = (-b + d) / (2.0*a);
    s2 = (-b - d) / (2.0*a);
    return 2;
  }
}


int main()
{
  // input
  std::cout << "Solve quadratic equation ax^2 + bx + c = 0 for\n";
  std::cout << "a =? ";
  double a;
  std::cin >> a;
  std::cout << "b =? ";
  double b;
  std::cin >> b;
  std::cout << "c =? ";
  double c;
  std::cin >> c;

  // computation
  std::complex<double> s1;
  std::complex<double> s2;

  const int n = solve_quadratic_equation (a, b, c, s1, s2);

  // output
  std::cout << "Number of solutions: " << n << "\n";
  std::cout << "Solutions:\n";
  if (n > 0) std::cout << s1 << "\n";
  if (n > 1) std::cout << s2 << "\n";

  return 0;

}
```

Solution to Exercise 141. We want to find all complex solutions to the equation

$$Ax^3 + Bx^2 + Cx + D = 0, \tag{B.3}$$

where $A, B, C, D \in \mathbb{C}$, and $A \neq 0$. The following method due to Scipione del Ferro and Tartaglia was published by Gerolamo Cardano in 1545. The method transforms the equation into an equivalent one that we can solve directly.

We first divide equation (B.3) by the leading coefficient A to arrive at an equivalent equation of the following form

$$x^3 + bx^2 + cx + d = 0, \tag{B.4}$$

where $b = B/A$, $c = C/A$, and $d = D/A$ are complex numbers.

Let us substitute $x = y - \frac{b}{3}$ into equation (B.4). Then the quadratic term disappears, and we get the equivalent equation

$$y^3 + 3qy - 2r = 0, \tag{B.5}$$

where

$$\begin{aligned} q &= \frac{3c - b^2}{9}, & \text{(B.6)} \\ r &= \frac{9bc - 27d - 2b^3}{54}. & \text{(B.7)} \end{aligned}$$

You are, of course, invited to check that for yourself. The left-hand side of (B.5) is called a *depressed* cubic. If $q = 0$, we are done already, since the solutions to $y^3 - 2r = 0$ are just the three complex third roots of of $2r$.

Let us assume for the remainder that $q \neq 0$. Then we further massage equation (B.5) by making the substitution $y = z - \frac{q}{z}$ and multiplying with z^3 on both sides of the equation. Then we arrive at

$$z^6 - 2rz^3 - q^3 = 0, \tag{B.8}$$

which can be viewed as a quadratic equation of the unknown variable z^3,

$$(z^3)^2 - 2r(z^3) - q^3 = 0. \tag{B.9}$$

We know how to solve (B.9) for z^3 (and then also for z), but before doing this, let's take a step back and see whether this really solves our problem: we know that x solves (B.3) if and only if $y = x + \frac{b}{3}$ solves (B.5). Moreover, let y and $z \neq 0$ be such that $y = z - \frac{q}{z}$. Then y solves (B.5) if and only if z solves (B.9). This means, every solution $z \neq 0$ to (B.9) gives us a solution $y = z - \frac{q}{z}$ to (B.5) and thus a solution $x = y - \frac{b}{3}$ to (B.3). Vice versa, every solution x to (B.3) gives us a solution $y = x + \frac{b}{3}$ to (B.5) and thus two nonzero solutions $z = \frac{y}{2} \pm \sqrt[2]{\frac{y^2}{4} + q}$ to (B.9), using $q \neq 0$.

Note: In writing $\sqrt[k]{c}$ for a complex number c, we choose one of the k roots arbitrarily, but at the same time we need to make sure that the choice does not matter. Indeed, the set of *two* numbers $\pm\sqrt[2]{\frac{y^2}{4} + q}$ does not depend on the particular choice of the square root.

To summarize, our original problem is solved by finding the (nonzero) solutions to (B.9), so let's turn to this latter problem.

We use the solution formula for quadratic equations in order to obtain

$$\begin{aligned} z^3 &= r \pm \sqrt[2]{r^2 + q^3}, & \text{(B.10)} \\ z &= \triangleright\sqrt[3]{r \pm \sqrt[2]{r^2 + q^3}}. & \text{(B.11)} \end{aligned}$$

Here, $\triangleright c$, for some number c, stands for one of the three numbers $c = c\xi_0, c\xi_1, c\xi_2$ where the ξ_i's are the third roots of unity, i.e. the three complex solutions to the equation $u^3 = 1$:

$$\xi_0 = 1, \quad \xi_1 = -\frac{1}{2} + \frac{\sqrt{3}}{2}i, \quad \xi_2 = -\frac{1}{2} - \frac{\sqrt{3}}{2}i.$$

Note that

$$\xi_i\xi_j = \xi_{i+j}, \tag{B.12}$$

where indices are taken modulo 3.

Equation (B.11) specifies a set of 6 values. Fixing our particular choices of roots, we can number them as

$$\begin{aligned} p_i &= \xi_i\sqrt[3]{r + \sqrt[2]{r^2 + q^3}}, \quad i = 0, 1, 2, && \text{(B.13)} \\ n_i &= \xi_i\sqrt[3]{r - \sqrt[2]{r^2 + q^3}}, \quad i = 0, 1, 2. && \text{(B.14)} \end{aligned}$$

We want to argue that we only need to consider the three solutions $z = p_i$ in order to get *all* the solutions $y = z - \frac{q}{z}$ to (B.5). For this, we first observe that the set $\{p_i n_i \mid i = 0, 1, 2\}$ is the set of third roots of $-q^3$. Indeed, by

$$p_i^3 n_i^3 = (r + \sqrt[2]{r^2 + q^3})(r - \sqrt[2]{r^2 + q^3}) = -q^3, \quad i = 0, 1, 2 \tag{B.15}$$

all three numbers have third power $-q^3$, and using (B.12) on top of (B.13) and (B.14), we can show that they are distinct:

$$p_i n_i = \xi_i^2\, p_0 n_0 = \xi_{2i}\, p_0 n_0, \quad i = 0, 1, 2 \ (\Rightarrow 2i = 0, 2, 1).$$

Consequently, there is an index $t \in \{0, 1, 2\}$ for which $p_t n_t = -q$, one of the third roots of $-q^3$. Having this, (B.13) and (B.14) together with (B.12) imply

$$p_t n_t = p_{t+1} n_{t+2} = p_{t+2} n_{t+1} = -q, \tag{B.16}$$

with indices taken modulo 3 again.

This in turn yields

$$\left(p_t - \frac{q}{p_t}, p_{t+1} - \frac{q}{p_{t+1}}, p_{t+2} - \frac{q}{p_{t+2}}\right) = \left(n_t - \frac{q}{n_t}, n_{t+2} - \frac{q}{n_{t+2}}, n_{t+1} - \frac{q}{n_{t+1}}\right), \tag{B.17}$$

meaning that in $y = z - \frac{q}{z}$, it does not matter whether we use $z = p_1, p_2, p_3$ or $z = n_1, n_2, n_3$.

That means we are done. We have found the three solutions

$$x_i = p_i - \frac{q}{p_i} - \frac{b}{3}, \quad i = 0, 1, 2$$

to equation (B.3), where p_i runs through the third roots of

$$r + \sqrt[2]{r^2 + q^3}$$

for $i = 0, 1, 2$.

The method described above is exactly the method implemented in the C++ program below.

```
// Prog: solve_cubic_equation.cpp
// computes the three complex solutions to a cubic equation
#include<iostream>
#include<complex>


// POST: return value is the number of distinct complex solutions
//       of the quadratic equation ax^2 + bx + c = 0. If there
//       are infinitely many solutions (a=b=c=0), the return
//       value is -1. Otherwise, the return value is a number n
//       from {0,1,2}, and the solutions are written to s1,..,sn
int solve_quadratic_equation (const std::complex<double> a,
                              const std::complex<double> b,
                              const std::complex<double> c,
                              std::complex<double>& s1,
                              std::complex<double>& s2)
{
  if (a == 0.0)
    // linear case: bx + c = 0
    if (b == 0.0)
      // trivial case: c = 0
      if (c == 0.0)
        return -1; // => infinitely many solutions
      else
        return 0;  // => no solution
    else {
      // bx + c = 0, b != 0   => one solution
      s1 = -c/b;
      return 1;
    }
  else {
    // ax^2 + bx + c = 0, a != 0 => two solutions
    const std::complex<double> d = std::sqrt(b*b-4.0*a*c);
    s1 = (-b + d) / (2.0*a);
    s2 = (-b - d) / (2.0*a);
    return 2;
  }
}

// POST: return value is the number of distinct complex solutions
//       of the cubic equation ax^3 + bx^2 + cx + d = 0. If there
//       are infinitely many solutions (a=b=c=d=0), the return
//       value is -1. Otherwise, the return value is a number n
//       from {0,1,2,3}, and the solutions are written to s1,..,sn
int solve_cubic_equation (const std::complex<double> a,
            const std::complex<double> b,
            const std::complex<double> c,
            const std::complex<double> d,
            std::complex<double>& s1,
            std::complex<double>& s2,
            std::complex<double>& s3)
{
  if (a == 0.0)
    // if a == 0 we can use the preceding function to solve
```

```
    // quadratic equations.
        return solve_quadratic_equation (b,c,d,s1,s2);
  else {
    // ax^3 + bx^2 + cx + d = 0, a != 0 => three solutions.
    // Some of the solutions may be equal due to multiple roots.

    // First we bring the equation  into the following form
    // x^3 + bn x^2 + cn x + dn = 0
    const std::complex<double> bn = b/a;
    const std::complex<double> cn = c/a;
    const std::complex<double> dn = d/a;

    // compute q and r
    const std::complex<double> q = (3.0*cn - bn*bn) / 9.0;
    const std::complex<double> r = (9.0*bn*cn - 27.0*dn - 2.0*bn*bn*bn) / 54.0;

    // define non-real root of unity
    const std::complex<double> rou(-0.5, std::sqrt(3.0)/2.0);
        // the variable temp is used to convert solutions y to x
    const std::complex<double> temp = bn/3.0;

    // check for the special case q == 0
    if (q == 0.0) {
      // compute        y, as in y^3 = 2r
      // substitution: x = y - b/3
      std::complex<double> y = std::pow(2.0*r, 1.0/3.0);

      // compute solutions for x
      s1 = y - temp;
      y *= rou; // rotate complex number y by -120 degrees
      s2 = y - temp;
      y *= rou; // rotate y by another -120 degrees
      s3 = y - temp;
    }
    else {
      // compute z, as in z^3 = r + sqrt(r^2 + q^3),
      // substitution 2: x = z - q/z - b/3
      std::complex<double> z = std::pow(r + std::sqrt(r*r + q*q*q), 1.0/3.0);

      // compute solutions for x
      s1 = z - q/z - temp;
      z *= rou; // rotate complex number z by -120 degrees
      s2 = z - q/z - temp;
      z *= rou; // rotate z by another -120 degrees
      s3 = z - q/z - temp;
    }

    return 3;
  }
}


// POST: Returns the magnitude of the complex number
// as^3 + bs^2 + cs + d. If s is an exact solution
// to the equation ax^3 + bx^2 + cx + d = 0 then this
// magnitude should be 0.
double check_solution(const std::complex<double> a,
            const std::complex<double> b,
            const std::complex<double> c,
            const std::complex<double> d,
            const std::complex<double> s) {

  // compute the ax^3 + bx^2 + cx + d and return its magnitude
  return std::abs(a*std::pow(s,3.0) + b*s*s + c*s + d);
}
```

```


int main()
{
  // input
  std::cout << "Solve cubic equation ax^3 + bx^2 + cx + d = 0 for\n";
  std::cout << "a =? ";
  std::complex<double> a;
  std::cin >> a;
  std::cout << "b =? ";
  std::complex<double> b;
  std::cin >> b;
  std::cout << "c =? ";
  std::complex<double> c;
  std::cin >> c;
  std::cout << "d =? ";
  std::complex<double> d;
  std::cin >> d;

  // computation
  std::complex<double> s1;
  std::complex<double> s2;
  std::complex<double> s3;

  const int n = solve_cubic_equation (a, b, c, d, s1, s2, s3);

  // output
  std::cout << "Number of solutions: " << n << "\n";
  std::cout << "Solutions:\n";
  if (n > 0) {
    std::cout << s1 << ", ";
    std::cout << "Error: " << check_solution(a,b,c,d,s1) << "\n";
  }
  if (n > 1){
    std::cout << s2 << ", ";
    std::cout << "Error: " << check_solution(a,b,c,d,s2) << "\n";
  }
  if (n > 2){
    std::cout << s3 << ", ";
    std::cout << "Error: " << check_solution(a,b,c,d,s3) << "\n";
  }

  return 0;

}
```