

B.10 Reference Types

Solution to Exercise 137.

- 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 138.

```
1  #include<iostream>
2
3  // POST: the values of i and j are swapped
4  void swap (int& i, int& j)
5  {
6      const 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 139. We implement the second version, the one that returns the normalization of r . This one has the advantage that it works for r values.

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 $|b|$.

```

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, const 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 (const int a, const 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     const 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 140.

```

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 (const double a, const 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 141.

- 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 142.

```

1 // Prog: solve_quadratic_equation.cpp
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  $s_1, \dots, s_n$ 
11 int solve_quadratic_equation (const std::complex<double> a,
12                             const std::complex<double> b,
13                             const 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         //  $bx + c = 0$ ,  $b \neq 0$  => one solution
27         s1 = -c/b;
28         return 1;
29     }
30     else {
31         const std::complex<double> d = std::sqrt(b*b-4.0*a*c);
32         if (d == 0.0) {
33             //  $ax^2 + bx + c = 0$ ,  $a \neq 0$ ,  $d=0$  => one solution
34             s1 = -b / (2.0*a);
35             return 1;
36         } else {
37             //  $ax^2 + bx + c = 0$ ,  $a \neq 0$ ,  $d \neq 0$  => two solutions
38             s1 = (-b + d) / (2.0*a);
39             s2 = (-b - d) / (2.0*a);
40             return 2;
41         }
42     }
43 }
44
45 // POST: solves the quadratic equation  $ax^2 + bx + c = 0$  and outputs
46 // the number of solutions as well as the solutions themselves
47 void test (const std::complex<double> a,
48           const std::complex<double> b,
49           const std::complex<double> c)
50 {
51     std::complex<double> s1;
52     std::complex<double> s2;
53
54     const int n = solve_quadratic_equation (a, b, c, s1, s2);
55
56     // output
57     std::cout << "Number of solutions: " << n << "\n";
58     std::cout << "Solutions:\n";
59     if (n > 0) std::cout << " " << s1 << "\n";
60     if (n > 1) std::cout << " " << s2 << "\n";
61 }
62
63 int main()
64 {
65     test (0, 0, 0);
66     //  $0 = 0$ : infinitely many solutions
67     test (0, 0, 2);
68     //  $2 = 0$ : no solution
69     test (0, 2, 2);
70     //  $2x + 2 = 0$ : one solution ( $x=-1$ )
71     test (2, 2, 2);
72     //  $2x^2 + 2x + 2 = 0$ : two solutions ( $x=-1/2 \pm \sqrt{3}/2 i$ )
73     test (1, 2, 1);
74     //  $x^2 + 2x + 1 = 0$ : one solution ( $x=-1$ )
75     test (std::complex<double>(0,1), 1, 1);
76     //  $ix^2 + x + 1 = 0$ : two solutions
77
78     return 0;
79 }

```

We asked you to test your function for the triples (a, b, c) from the set

$\{(0, 0, 0), (0, 0, 2), (0, 2, 2), (2, 2, 2), (1, 2, 1), (i, 1, 1)\}$.

The output to these inputs should be:

- Input: $(0, 0, 0)$, number of solutions: infinitely many (-1)
- Input: $(0, 0, 2)$, number of solutions: 0
- Input: $(0, 2, 2)$, number of solutions: 1,
solutions: $s1 == (-1, 0)$
- Input: $(2, 2, 2)$, number of solutions: 2,
solutions: $s1 == (-0.5, 0.866025)$, $s2 == (-0.5, -0.866025)$
- Input: $(1, 2, 1)$, number of solutions: 1,
solutions: $s1 == (-1, 0)$
- Input: $(i, 1, 1)$, number of solutions: 2,
solutions: $s1 == (-0.624811, -0.300243)$, $s2 == (0.624811, 1.30024)$

Solution to Exercise 143. We want to find all complex solutions to the equation

$$Ax^3 + Bx^2 + Cx + D = 0, \quad (\text{B.3})$$

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

$$x^3 + bx^2 + cx + d = 0, \quad (\text{B.4})$$

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

$$y^3 + 3qy - 2r = 0, \quad (\text{B.5})$$

where

$$q = \frac{3c - b^2}{9}, \quad (\text{B.6})$$

$$r = \frac{9bc - 27d - 2b^3}{54}. \quad (\text{B.7})$$

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 $2r$.

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

$$z^6 - 2rz^3 - q^3 = 0, \quad (\text{B.8})$$

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

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

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 $z = \frac{y}{2} \pm \sqrt{\frac{y^2}{4} + q}$ 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 $\pm \sqrt{\frac{y^2}{4} + q}$ 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

$$z^3 = r \pm \sqrt{r^2 + q^3}, \quad (\text{B.10})$$

$$z = \triangleright \sqrt[3]{r \pm \sqrt{r^2 + q^3}}. \quad (\text{B.11})$$

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

$$\xi_i \xi_j = \xi_{i+j}, \quad (\text{B.12})$$

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

$$p_i = \xi_i \sqrt[3]{r + \sqrt{r^2 + q^3}}, \quad i = 0, 1, 2, \quad (\text{B.13})$$

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


```

14                                     const std::complex<double> c,
15                                     std::complex<double>& s1,
16                                     std::complex<double>& s2)
17 {
18     if (a == 0.0)
19         // linear case:  $bx + c = 0$ 
20         if (b == 0.0)
21             // trivial case:  $c = 0$ 
22             if (c == 0.0)
23                 return -1; // => infinitely many solutions
24             else
25                 return 0; // => no solution
26         else {
27             //  $bx + c = 0$ ,  $b \neq 0$  => one solution
28             s1 = -c/b;
29             return 1;
30         }
31     else {
32         //  $ax^2 + bx + c = 0$ ,  $a \neq 0$  => two solutions
33         const std::complex<double> d = std::sqrt(b*b-4.0*a*c);
34         s1 = (-b + d) / (2.0*a);
35         s2 = (-b - d) / (2.0*a);
36         return 2;
37     }
38 }
39
40 // POST: return value is the number of distinct complex solutions
41 //       of the cubic equation  $ax^3 + bx^2 + cx + d = 0$ . If there
42 //       are infinitely many solutions ( $a=b=c=d=0$ ), the return
43 //       value is -1. Otherwise, the return value is a number  $n$ 
44 //       from  $\{0,1,2,3\}$ , and the solutions are written to  $s1, \dots, s_n$ 
45 int solve_cubic_equation (const std::complex<double> a,
46                          const std::complex<double> b,
47                          const std::complex<double> c,
48                          const std::complex<double> d,
49                          std::complex<double>& s1,
50                          std::complex<double>& s2,
51                          std::complex<double>& s3)
52 {
53     if (a == 0.0)
54         // if  $a = 0$  we can use the preceding function to solve
55         // quadratic equations.
56         return solve_quadratic_equation (b,c,d,s1,s2);
57     else {
58         //  $ax^3 + bx^2 + cx + d = 0$ ,  $a \neq 0$  => three solutions.
59         // Some of the solutions may be equal due to multiple roots.
60
61         // First we bring the equation into the following form
62         //  $x^3 + b_n x^2 + c_n x + d_n = 0$ 
63         const std::complex<double> bn = b/a;
64         const std::complex<double> cn = c/a;
65         const std::complex<double> dn = d/a;
66
67         // compute  $q$  and  $r$ 
68         const std::complex<double> q = (3.0*cn - bn*bn) / 9.0;
69         const std::complex<double> r = (9.0*bn*cn - 27.0*dn - 2.0*bn*bn*bn) / 54.0;
70
71         // define non-real root of unity
72         const std::complex<double> rou(-0.5, std::sqrt(3.0)/2.0);
73         // the variable temp is used to convert solutions  $y$  to  $x$ 
74         const std::complex<double> temp = bn/3.0;
75
76         // check for the special case  $q = 0$ 
77         if (q == 0.0) {
78             // compute  $y$ , as in  $y^3 = 2r$ 

```

```

79     // substitution:  $x = y - b/3$ 
80     std::complex<double> y = std::pow(2.0*r, 1.0/3.0);
81
82     // compute solutions for x
83     s1 = y - temp;
84     y *= rou; // rotate complex number y by -120 degrees
85     s2 = y - temp;
86     y *= rou; // rotate y by another -120 degrees
87     s3 = y - temp;
88 }
89 else {
90     // compute z, as in  $z^3 = r + \sqrt{r^2 + q^3}$ ,
91     // substitution 2:  $x = z - q/z - b/3$ 
92     std::complex<double> z = std::pow(r + std::sqrt(r*r + q*q*q), 1.0/3.0);
93
94     // compute solutions for x
95     s1 = z - q/z - temp;
96     z *= rou; // rotate complex number z by -120 degrees
97     s2 = z - q/z - temp;
98     z *= rou; // rotate z by another -120 degrees
99     s3 = z - q/z - temp;
100 }
101
102 return 3;
103 }
104 }
105
106
107 // POST: Returns the magnitude of the complex number
108 // as  $ax^3 + bx^2 + cx + d$ . If s is an exact solution
109 // to the equation  $ax^3 + bx^2 + cx + d = 0$  then this
110 // magnitude should be 0.
111 double check_solution(const std::complex<double> a,
112                     const std::complex<double> b,
113                     const std::complex<double> c,
114                     const std::complex<double> d,
115                     const std::complex<double> s) {
116
117     // compute the  $ax^3 + bx^2 + cx + d$  and return its magnitude
118     return std::abs(a*std::pow(s,3.0) + b*s*s + c*s + d);
119 }
120
121
122 int main()
123 {
124     // input
125     std::cout << "Solve cubic equation  $ax^3 + bx^2 + cx + d = 0$  for\n";
126     std::cout << "a =? ";
127     std::complex<double> a;
128     std::cin >> a;
129     std::cout << "b =? ";
130     std::complex<double> b;
131     std::cin >> b;
132     std::cout << "c =? ";
133     std::complex<double> c;
134     std::cin >> c;
135     std::cout << "d =? ";
136     std::complex<double> d;
137     std::cin >> d;
138
139     // computation
140     std::complex<double> s1;
141     std::complex<double> s2;
142     std::complex<double> s3;
143

```

```
144     const int n = solve_cubic_equation (a, b, c, d, s1, s2, s3);
145
146     // output
147     std::cout << "Number of solutions: " << n << "\n";
148     std::cout << "Solutions:\n";
149     if (n > 0) {
150         std::cout << s1 << ", ";
151         std::cout << "Error: " << check_solution(a,b,c,d,s1) << "\n";
152     }
153     if (n > 1){
154         std::cout << s2 << ", ";
155         std::cout << "Error: " << check_solution(a,b,c,d,s2) << "\n";
156     }
157     if (n > 2){
158         std::cout << s3 << ", ";
159         std::cout << "Error: " << check_solution(a,b,c,d,s3) << "\n";
160     }
161
162     return 0;
163
164 }
```
