Appendix B

Solutions

B.1 A first C++ program

Solution to Exercise 1. (d) and (f) are not identifiers, since they do not start with a letter. (g) is not an identifier, since it contains the character #. (b) is not allowed as a variable name, but it is a valid identifier.

Solution to Exercise 2. (c) is not an expression, since the first operand of the assignment operator must be an lvalue, but `1` is a literal, hence an rvalue. (f) is not an expression, since there is no closing parenthesis for the opening one. (h) is invalid, since `(a*3)` is an rvalue, but the left operand of the assignment operator must be an lvalue.

Solution to Exercise 3. (a), (e), and (g) are rvalues by definition of the binary multiplication operator. (b) and (d) are lvalues by definition of the assignment operator.

Solution to Exercise 4. (a) has value 6, obtained by multiplying the value of the primary expression `1` with the value of the composite expression `(2*3)`. The latter value is 6, for the same reason. (b) has value 5, obtained by assigning value 5 to `b` first (right assignment), and then to `a` (left assignment). (d) has value 1, by definition of the assignment operator. (e) has value 35, since the operands `(a=5)` and `(b=7)` have values 5 and 7, respectively.

In case of (g), the value is unspecified. If the right operand is evaluated first, we get value 25, but if the left operand comes first, `b` may have some value other than 5, and the left operand evaluates to this other value. The final result will not be 25, then.

Solution to Exercise 5. b) is incorrect, since `std::cin >> a` tries to change the value of the constant `a`. c) is incorrect, since there is an uninitialized constant `a`. e) is incorrect, since the assignment `a = 6` tries to change the value of the constant `a`. Same thing with f): although the value will not be changed, it is incorrect. The exact wording was that it is impossible to store anything under the address of a constant, even if it is the same value as before. The correct programs are thus a), d), g). Among them, only d) violates the **Const Guideline**, since c's value never changes, but yet c is not defined as a constant.

Solution to Exercise 6. The least common multiple of $2, \ldots, n$ is the product of all *maximal* prime powers less or equal to n. For $n = 10$, for example, we get $2^3 \cdot 3^2 \cdot 5 \cdot 7 = 2520$. For $n = 20$, we get

$$2^4 \cdot 3^2 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 = 232792560,$$

and for $n = 30$

$$2^4 \cdot 3^3 \cdot 5^2 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29 = 2329089562800.$$

The following program computes these numbers. Interestingly, declaring `lcm` of type `int` leads to an incorrect value for $n = 30$ on a 32-bit system, due to overflow. But using `ifm::integer`, our arbitrary-precision integers also mentioned in Challenge 11, yields the correct results.

```
// Prog: lcm.cpp (least common multiple)
// computes the least common multiple of 10, 20, and 30

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

int main()
{
  // the least common multiple of the number 2,...,n is the product
  // of all maximal prime powers that are less or equal to n.
  // Example: the least common multiple of 2,...,10 is 2^3 * 3^2 * 5 * 7

  std::cout << "Least common multiple of 2 up to...\n";
  // 10: 2^3 * 3^2 * 5 * 7
  ifm::integer lcm = 2*2*2 * 3*3 * 5 * 7;
  std::cout << 10 << ": " << lcm << "\n";

  // 20: 2^4 * 3^2 * 5 * 7 * 11 * 13 * 17 * 19
  lcm = lcm * 2 * 11 * 13 * 17 * 19;
  std::cout << 20 << ": " << lcm << "\n";

  // 30: 2^4 * 3^3 * 5^2 * 7 * 11 * 13 * 17 * 19 * 23 * 29
  lcm = lcm * 3 * 5 *23 * 29;
  std::cout << 30 << ": " << lcm << "\n";

  return 0;

}
```

Solution to Exercise 7.

```
// Program: multhree.cpp
// Compute the product of three numbers.

#include <iostream>

int main()
```

```
{
  // input of a, b and c
  std::cout << "Compute a * b * c for a =? ";
  int a;
  std::cin >> a;

  std::cout << "... and b =? ";
  int b;
  std::cin >> b;

  std::cout << "... and c =? ";
  int c;
  std::cin >> c;

  // output a * b * c,
  std::cout << a << " * " << b  << " * " << c << " = "
            << a * b * c << ".\n";
  return 0;
}
```

Solution to Exercise 8.

```
// Program: power20.cpp
// Raise a number to the power twenty.

#include <iostream>

int main()
{
  // input
  std::cout << "Compute a^20 for a =? ";
  int a;
  std::cin >> a;

  // computation
  const int b = a * a; // b = a^2
  const int c = b * b; // c = a^4
  const int d = c * c; // d = a^8
  const int e = d * d; // e = a^16

  // output e * c, i.e. a^20
  std::cout << a << "^20 = " << e * c << ".\n";
  return 0;
}
```

Solution to Exercise 9. Here is the well-formatted program, complete with informative output and sensible comments. We have also fixed the two errors (`main()` instead of `main[]`, and `std::cin >> b` instead of `cin >> b`). This solves parts a), b), d), and e).

```
// SquareProduct.cpp
// Reads in two numbers a and b and outputs (a*b)^2,
// the square of their product
#include <iostream>

int main() {

   // input
   std::cout << "a =? ";
   int a;
```

```
    std::cin >> a;

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

    // computation and output
    const int c = a * b;
    std::cout << "(a*b)^2 = ";
    std::cout << c * c << ".\n";

    return 0;
}
```

For part c), here is the list of composite expressions from the original (fixed) program, along with their status.

- `std::cin >> a` (lvalue)
- `std::cin >> b` (lvalue)
- `a * b` (rvalue)
- `c = a * b` (lvalue)
- `c * c` (rvalue)
- `std::cout << c * c` (lvalue)

Solution to Exercise 10.

```
// Prog: age_verification.cpp
// outputs the age group that is not allowed to buy alcohol

#include<iostream>

int main()
{
  const int year = 2009;
  std::cout << "No alcohol to people born in the years "
            << year-17 << " - " << year << "!\n";
  std::cout << "For people born in " << year-18
            << ", check the id!\n";

  return 0;
}
```

Solution to Exercise 11. Here are the two programs:

```
// Program: power8_slow.cpp
// Raise a number to the eighth power,
// using integers of arbitrary size
// and with seven multiplications

#include <iostream>
#include <IFM/integer.h>
```

```

int main()
{
  // input (no prompt, as we intend to read from file)
  ifm::integer a;
  std::cin >> a;

  // computation
  ifm::integer b = a * a; // b = a^2
  b = b * a;              // b = a^3
  b = b * a;              // b = a^4
  b = b * a;              // b = a^5
  b = b * a;              // b = a^6
  b = b * a;              // b = a^7
  b = b * a;              // b = a^8

  // no output, as we are interested in computation time
  return 0;
}
```

```
// Program: power8_fast.cpp
// Raise a number to the eighth power,
// using integers of arbitrary size
// and with three multiplications

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

int main()
{
  // input (no prompt, as we intend to read from file)
  ifm::integer a;
  std::cin >> a;

  // computation
  ifm::integer b = a * a; // b = a^2
  b = b * b;              // b = a^4
  b = b * b;              // b = a^8

  // no output, as we are interested in computation time
  return 0;
}
```

Running them on inputs up to 100,000 decimal digits, you will see that `power8_fast.cpp` is indeed faster. But while the number of multiplications performed by `power8_fast.cpp` is only 42% of the corresponding number for `power8_slow.cpp` (3 vs. 7), this does not directly translate to the runtimes. Instead, you will observe that `power8_fast.cpp` needs around 75% of the time required by `power8_slow.cpp` (and this remains stable as the inputs get larger). Thus the speedup is much less than you might have expected from just counting multiplications. Why is this so?

In order to really answer this, you would have to know how the type `ifm::integer` is implemented; but for our discussion, it is enough to know that multiplication of two `ifm::integer`s works according to the school method. Do you remember how this is done? When you multiply two numbers on a piece of paper, you multiply the first number with each individual digit of the second number and write down all the results (properly aligned). Then you just add them up (see Page 11 for an example of the school method).

If the two numbers have m and n digits, respectively, you have n intermediate results, with m or $m+1$ digits each, meaning that in total, you will write down roughly mn digits. This is also what the computer does, and the time to do it will be roughly proportional to mn. Let us cheat a little and assume that the time is really mn (it could in reality be around $10mn$, or any other factor times mn, but the speedup of `power8_fast.cpp` compared to `power8_slow.cpp` does not depend on this).

With this knowledge at hand, we can already understand the 75% from above. Let's look at `power8_slow.cpp` first, and let us assume that the input number `a` has m digits. Then the first multiplication (`a * a`) needs time m^2 and results in the number `b` with (roughly) $2m$ digits. The next multiplication `b * a` features numbers with $2m$ and m digits and therefore takes time $2m^2$. The result has (roughly) $3m$ digits. Then we multiply numbers with $3m$ and m digits, in time $3m^2$, and the result has (roughly) $4m$ digits. Continuing in this way, we see that the time to perform all multiplications is (roughly) $(1+2+\cdots+7)m^2 = 28m^2$.

Now comes `power8_fast.cpp`. The first multiplication again takes time m^2, but since the second one features two numbers with (roughly) $2m$ digits, the time for the second multiplication is $4m^2$ and results in a number with (roughly) $4m$ digits. In the third multiplication, we therefore multiply two $4m$-digit numbers, in time $16m^2$. In total, this takes time $(1+4+16)m^2 = 21m^2$. And since 21 is 75% of 28, this calculation is a pretty good explanation of the experimental observations.

Solution to Exercise 12. For the lower bound in a), we argue by induction that $a_i \leq a^{2^i}$ for all i (we can't do more than double the number in each step). In order to get $a_t = n$, we therefore must have

$$a^{2^t} \geq a_t = a^n,$$

or $2^t \geq n$. It follows that

$$t \geq \lg n \geq \lfloor \lg n \rfloor = \lambda(n).$$

For the upper bound, we have to come up with a computation for a^n that needs at most $\lambda(n) + \nu(n) - 1$ steps. This is simple (and called the binary method). By doubling a $\lambda(n)$ times, we can compute in $\lambda(n)$ steps all powers of the form a^{2^i} that are less or equal to a^n. For example, in $\lambda(20) = 4$ steps, we can get the values a, a^2, a^4, a^8, a^{16}. Since n is the sum of exactly $\nu(n)$ of these 2^i, a^n is the *product* of exactly $\nu(n)$ of these a^{2^i} (this is a simple consequence of the formula $a^{n+m} = a^n \cdot a^m$). It follows that we can obtain a^n by simply multiplying these $\nu(n)$ values together, and since we already have them, this can be done in $\nu(n) - 1$ further multiplications.

For b), we give an example where the upper bound is not tight. Consider $n = 15$ (1111 in binary). We have $\lambda(15) = 3$ and $\nu(15) = 4$, so the binary method would need 6 multiplications. But we can do it with five multiplications, as follows:

```
a1 = a0 * a0   // a^2
a2 = a1 * a0   // a^3
a3 = a2 * a2   // a^6
a4 = a3 * a3   // a^12
a5 = a4 * a2   // a^15
```

In general, no exact formula for $\ell(n)$ is known.