Solution to Exercise 33. This program contains four errors.

- line 1: `iostraem` $\longrightarrow$ `iostream`
- line 4: `unsinged` $\longrightarrow$ `unsigned`
- line 6: semicolon missing after *condition* of `for`-statement
- line 6: superfluous opening curly brace `{`

Things that are ok are in particular the following.

- The `+1` in line 4 is a valid expression of type `int` (and value 1), constructed from the unary addition operator and the literal operand `1`. In the assignment, this value is converted to the type `unsigned int`.
- `{ std::cin >> x; }` The variable `x` here is the one defined in line 4, even though we have put the input statement into an extra scope.
- The *init-statement* of the `for`-loop is ok, since we may assign an `unsigned int` value to an `int` variable.

The fixed program looks like this.

```
 1 #include<iostream>
 2
 3 int main()
 4 {
 5   unsigned int x = +1;
 6   { std::cin >> x; }
 7   for (int y=0u; y < x;)
 8     std::cout << ++y;
 9   return 0;
10 }
```

It outputs the sequence of positive natural numbers smaller or equal to the input number (without any blanks, this will be difficult to read, though).

Solution to Exercise 34. There are two problems in the code. First, the declarative region of the declaration of `i` in line 3 is the body of the loop only. Therefore, the expression `++i < 10` in line 5 is undefined. Moreover, even if the declaration would extend up to line 5, the loop would be infinite because `i` is initialized with 1 in every iteration. Thus, the condition `++i < 10` will never be fulfilled. This problem can be fixed by moving the declaration of `i` out of the loop.

Second, as the variable `s` is initialized with `0`, none of the multiplications in the loop changes the value of `s`. This problem can be fixed by initializing `s` with `1`. The corrected program then looks as shown below.

```
1    unsigned int s = 1;
2    int i = 1;
3    do {
4      if (i % 2 == 1) s *= i;
5    } while (++i < 10);
```

This computes the product of the odd numbers $1, 3, \ldots, 9$.

Solution to Exercise 35.

declaration in line	declarative region	scope
4	3–20	4–11, 15–20
6	5–17	6–17
10	8–14	10–14
12	8–14	12–14
15	5–17	15–17
18	3–20	18–20

The output of the program is 4. There is only one iteration of the loop, and that one sets the s relevant in line 16 to 2.

Solution to Exercise 36.

- For input -1: no iteration, output is 0.
- For input 1:

begin of iteration	s	x	i
1	0	1	0
2	0	1	1

  No second iteration, output is 0.

- For input 2:

begin of iteration	s	x	i
1	0	2	0
2	0	2	1
3	1	2	2

  No third iteration, output is 1.

- For input 3:

begin of iteration	s	x	i
1	0	3	0
2	0	3	1
3	1	3	2
4	3	4	3
5	6	7	4

We see that x grows faster than i, so the condition `i < x` will mathematically never be satisfied. Due to overflow, it might eventually be satisfied in practice, but the output is undefined.

Solution to Exercise 37. Here are the problems.

- line 5: x is undeclared, because the x in line 4 is local to the block in line 4.
- line 5: it should be `std::cin >> x` and not `std::cin << x`.
- line 7: this is an infinite loop: since y is of type `unsigned int`, its value will always be nonnegative. The condition `y >= 0` is therefore always satisfied, regardless of the way y is changed in the loop.
- line 9: s appears outside of its scope (lines 7-8), so it is undefined here.

Solution to Exercise 38. Let x and y be the values of x and y. For $x = 0$, we get an infinite loop, since y is initialized with 1 but never changes. For $x \geq 1$, y is initialized with $x + 1$ and decreased by x in the subsequent iterations. Unless $x = 1$, y therefore becomes "negative" in the second iteration, leading to a not well-defined positive value of y. For $x = 1$, the loop terminates after two iterations and outputs value $2 + 1 = 3$.

To summarize, the behavior is well-defined exactly for the following input/output pair.

input	output
1	3

Solution to Exercise 39. Let's analyze what the code does. In the do-statement, k is incremented (starting from 0), until it reaches the value n of n. After this, e is set to *false*, and the loop terminates. Therefore, k increases n times, meaning that x (initially of value 1) is multiplied by 2^n.

The output is therefore 2^n, for input n. Here is a more readable version of this code.

```
1    unsigned int n;
2    std::cin >> n;
3
4    unsigned int x = 1;
5    for (int k = 0; k < n; ++k)
```

```
6       x *= 2;
7
8     std::cout << x;
```

Solution to Exercise 40. Let's first see what the inner `for`-loop does. For given value `i` of `i`, `j` starts from 1 and gets incremented by 1 during the `if`-statement's condition. The inner loop terminates with `break` if the value *before* the increment reaches `i`. This means, `s += j - 1` is executed for `j` running from 2 through `i`. The inner loop therefore increments `s` by

$$\sum_{k=1}^{i-1} k = \frac{i(i-1)}{2}, \quad i \geq 2.$$

The other `do`-loop increments `i` (starting from -10) by 1, as long as the incremented `i` is smaller or equal to `x`. The inner loop is therefore executed for the values $i = -10, -9, \ldots, x$, and we get

$$s = \sum_{i=2}^{x} \frac{i(i-1)}{2}.$$

A more readable version is therefore

```
1     #include <iostream>
2     int main()
3     {
4       int x;
5       std::cin >> x;
6       int s = 0;
7       for (int i = 2; i <= x; ++i)
8         s += i * (i-1) / 2;
9       std::cout << s << "\n";
10      return 0;
11    }
```

Solution to Exercise 41.

```
1   // Prog: fak-1.C
2   // compute the n!
3
4   #include <iostream>
5
6   int main ()
7   {
8     std::cout << "Factorial of n =? ";
9     int n;
10    std::cin >> n;
11
12    // computation
```

```
13    unsigned int fak = 1;
14    for (unsigned int i = 2; i <= n; ++i) fak *= i;
15
16    // output
17    std::cout << "Factorial of " << n << " is " << fak << ".\n";
18
19    return 0;
20  }
```

Solution to Exercise 42.

```
1   // Program: dec2bin.C
2   // Output reverse binary representation of a number n
3
4   #include <iostream>
5
6   int main()
7   {
8     // input
9     std::cout << "Number n =? ";
10    unsigned int n;
11    std::cin >> n;
12
13    // computation and output
14    std::cout << "Binary representation is: ";
15    do { // a while loop cannot correctly handle n == 0
16      std::cout << n % 2;
17      n /= 2;
18    } while (n > 0);
19    std::cout << "\n";
20
21    return 0;
22  }
```

Solution to Exercise 43.

```
1   // Prog: cross_sum.C
2   // compute the cross sum of a natural number
3
4   #include <iostream>
5
6   int main ()
7   {
8     std::cout << "Cross sum of n =? ";
9     unsigned int n;
10    std::cin >> n;
11
12    // computation
13    unsigned int cross = 0;
14    for (unsigned int m = n; m > 0; m /= 10) cross += m % 10;
15
16    // output
17    std::cout << "Cross sum of " << n << " is "
18              << cross << ".\n";
19
20    return 0;
21  }
```

Solution to Exercise 44.

```
1  // Program: perfect.C
2  // Find all perfect numbers up to an input number n
3
4  #include <iostream>
5
6  int main()
7  {
8    // input
9    std::cout << "Find perfect numbers up to n =? ";
10   unsigned int n;
11   std::cin >> n;
12
13   // computation and output
14   std::cout << "The following numbers are perfect.\n";
15   for (unsigned int i = 1; i <= n ; ++i) {
16     // check whether i is perfect
17     unsigned int sum = 0;
18     for (unsigned int d = 1; d < i; ++d)
19       if (i % d == 0) sum += d;
20     if (sum == i)
21       std::cout << i << " ";
22   }
23   std::cout << "\n";
24
25   return 0;
26 }
```

The program output looks as follows.

```
Find perfect numbers from 1 to n =? 50000
The following numbers are perfect.
6 28 496 8128
```

Solution to Exercise 45. We start as in Exercise 42, but instead of outputting the digits (which would then appear in reverse order), we use them to build up a number m whose binary representation is the *reverse* of the binary representation of n. Then we proceed again as in Exercise 42, but this time starting with m. We have to be careful not to lose leading zeros in the binary representation of m, since we want to output them as well. Therefore we also need to count the number of binary digits in n.

```
1  // Program: dec2bin2.C
2  // Output binary representation of a number n
3
4  #include <iostream>
5
6  int main()
7  {
8    // input
9    std::cout << "Number n =? ";
10   unsigned int n;
11   std::cin >> n;
12
13   // compute / count digits and build up number m with
14   // reverse representation
15   unsigned int digits = 0;
16   unsigned int m = 0;
17   do { // a while loop cannot correctly handle n == 0
18     ++digits;
```

```
19      m = m * 2 + n % 2;
20      n /= 2;
21    } while (n > 0);
22
23    // now process m as in dec2bin.C
24    std::cout << "Inverse binary representation is: ";
25    for (int d = 0; d < digits; ++d) {
26      std::cout << m % 2;
27      m /= 2;
28    }
29    std::cout << "\n";
30
31    return 0;
32  }
```

Some people suggested that the problem can be solved with just one loop, as follows:

```
unsigned int s = 0;
for ( unsigned int k=1; n>0; n/=2 ){
  s += k * (n % 2);
  k *= 10;
}
std::cout << s;
```

This performs the following "hack": it builds up a *decimal* number s whose digits are just 0's and 1's. These digits are in fact exactly the digits of n's binary representation: the digits of n are computed in reverse order and are successively *prepended* to s. This means that in s, they appear in the right order.

The hack is that we can save the output loop as in `dec2bin2.C` above, by misusing standard integer output to do it for us. This is a very nice solution, but it has one drawback: the number s is substantially larger than n: where n needs k binary digits, s needs k *decimal* digits. This implies that (on a 32-bit system) only numbers n up to roughly $2^{10} = 1024$ can be processed, since beyond that, we get an overflow in s.

Solution to Exercise 46.

```
1   // Prog: dice.C
2   // Pete rolls three four-sided dice, and Colin rolls 2 six-sided dice.
3   // What is the probability that Pete / Colin / nobody has the higher
4   // total number of points?
5   #include<iostream>
6   #include<cassert>
7
8   int main()
9   {
10    unsigned int pete_throws =  4*4*4; // number of possibilities for Pete
11    unsigned int colin_throws = 6*6;  // number of possibilities for Colin
12    unsigned int pairs_of_throws = pete_throws * colin_throws;
13
14    unsigned int pete_wins = 0; // pairs of throws won by Pete
15    unsigned int colin_wins = 0; // pairs of throws won by Colin
16    unsigned int draws = 0; // pairs of throws that are a draw
17
18    // now do the following: for each possible pair of throws
19    // (all of them have the same probability of occurring),
20    // compute the scores of both players and determine the
```

```
21    // winner (or that the throw is a draw)
22    for (unsigned int pete=0; pete < pete_throws; ++pete) {
23      // compute score of Pete, interpreting pete as a throw, i.e.
24      // a base-4 number with three digits
25      // (digit value i encodes i+1 points, i=0,1,2,3)
26      unsigned int pete_score = 0;
27      unsigned int p = pete;
28      for (int i=0; i<3; ++i) {
29        pete_score += p % 4 + 1;
30        p /= 4;
31      }
32      for (unsigned int colin=0; colin < colin_throws; ++colin) {
33        // compute score of Colin, interpreting colin as a throw, i.e.
34        // base-6 number with two digits
35        // (digit value i encodes i+1 points, i=0,1,2,3,4,5)
36        unsigned int colin_score = 0;
37        unsigned int c = colin;
38        for (int i=0; i<2; ++i) {
39          colin_score += c % 6 + 1;
40          c /= 6;
41        }
42        // who wins in this pair of throws?
43        if (pete_score >  colin_score) ++pete_wins;
44        if (pete_score <  colin_score) ++colin_wins;
45        if (pete_score == colin_score) ++draws;
46      }
47    }
48    // compute the probabilities
49    std::cout << "Pete wins with probability   "
50              << pete_wins << "/" << pairs_of_throws << ".\n";
51    std::cout << "Colin wins with probability  "
52              << colin_wins << "/" << pairs_of_throws << ".\n";
53    std::cout << "Draw occurs with probability "
54              << draws << "/" << pairs_of_throws << ".\n";
55
56    return 0;
57  }
```

The output is

```
Pete wins with probability   1152/2304.
Colin wins with probability  870/2304.
Draw occurs with probability 282/2304.
```

After cancellation, these numbers are $1/2$, $145/384$, and $47/384$. The game is therefore not fair, since Pete is much better off.

Solution to Exercise 47.

```
1   // Program: mersenne.C
2   // computes a nontrivial factor of 2^67-1
3
4   #include <iostream>
5   #include "integer.h"
6
7   int main()
8   {
9     // build up 2^67-1
10    ifm::integer p = 1;
11    for (int i=0; i<67; ++i) p *= 2;
12    --p;
```

```
13
14    std::cout << "Factoring 2^67-1 = " << p << "...\n";
15
16    // now try all potential divisors (odd numbers)
17    for (ifm::integer d = 3; d < p; d += 2)
18      if (p % d == 0) {
19        std::cout << p << " = " << d << " * " << p / d << ".\n";
20        break;
21      }
22
23    return 0;
24  }
```

Solution to Exercise 48. Here is the idea: we build up the placements columnwise, where we represent a partial placement in the first $column$ columns by a $column$-digit number $placement$ in base n; digit i encodes the position of the queen in column i (from 0 (bottommost) to $n-1$ (topmost)).

In fact, we represent $placement$ by a value of type `unsigned int`; the digits in base n can be accessed through repeated modulus and integral division operations with second operand n. The algorithm iteratively enumerates all threat-free partial placements. Whenever the partial placement is full, we increse a $solution$ counter.

In the beginning of each iteration, we are given a partial placement $placement$ up to column $column$, along with the information whether it is threat-free (in the beginning ($column = 0$), the partial placement is empty ($placement = 0$) and obviously threat-free). Moreover, we will maintain the invariant that $placement$ restricted to columns $\{1, \ldots, c\}, c < column$ is threat-free.

If the whole $placement$ is threat-free, we extend it to the next column, where we start with the queen in the bottommost position. The invariant still holds in this case. The extension is done through a multiplication of $placement$ by n. In the case where we have a threat in $placement$, we switch to the next possible partial placement within our current set of columns. This can simply be done by adding one to $placement$; the least significant base-n digits that have become 0 now correspond to columns where all possible queen positions 0 through $n-1$ have been tried already. Therefore we remove these columns from our partial placement and continue with a shorter partial placement up to the last column where not everything has been tried yet. Since this new $placement$ agrees with the old one except in its last column, we know that our invariant also holds in this case.

After updating $placement$ that way, we now have a new partial placement for which we have to check whether it's threat-free. By the invariant we know that the partial placement excluding the last column is threat-free, so we only have to check for threats with the queen in the last column of our partial placement. This gives us the information whether the current $placement$ is threat-free, and we start all over again with the next iteration. Here is the program.

```
1  // Program: n_queens.C
2  // compute number of solutions to the n-queens puzzle
3  // for given input number n (for 32-bit systems, this
```

```
4  // works up to n = 9 only); using the nonstandard g++
5  // type "unsigned long long" instead of "unsigned int"
6  // everywhere, the program works for larger values (but
7  // becomes very slow around n=14)
8
9  #include<iostream>
10
11 int main ()
12 {
13   // input
14   std::cout << "Solve n-queens puzzle for n =? ";
15   unsigned int n;
16   std::cin >> n;
17
18   unsigned int solutions = 0; // number of solutions
19   unsigned int placement = 0; // partial placement (base n)...
20   unsigned int column = 0;    // ...up to this column
21   bool threat_free = true;    // is the placement threat-free?
22
23   for(;;) {
24     if (threat_free && column < n) {
25       // extend placement to next column
26       placement *= n;
27       ++column;
28     } else {
29       if (threat_free) ++solutions; // n == column
30       // try next placement up to this column
31       placement += 1;
32       while (placement % n == 0) {
33         // column exhausted, remove from placement
34         placement /= n;
35         --column;
36       }
37       if (column == 0) break; // everything tried, stop
38     }
39     // check whether placement is threat-free
40     threat_free = false;                    // we don't know yet
41     unsigned int q_column = placement % n; // queen in column
42     unsigned int c = column-1;              // column to the left
43     unsigned int p_c = placement / n;       // placement up to c
44     while (c > 0) {
45       unsigned int q_c = p_c % n;           // queen in c
46       // is q_column threatened by q_c?
47       if (q_c == q_column) break;           // horizontal threat
48       if (q_c - q_column == column - c) break; // diagonal threat 1
49       if (q_column - q_c == column - c) break; // diagonal threat 2
50       --c; p_c /= n; // go one column left
51     }
52     if (c == 0) threat_free = true;
53   }
54
55   std::cout << solutions << " solutions\n";
56
57   return 0;
58 }
```

Solution to Exercise 49.

```
1  // Prog: famous_last_digits.C
2  // outputs the last 10 decimal digits of Mersenne prime 2^{43,112,609}-1
3
4  #include<iostream>
5  #include "integer.h"
```

```
 6
 7  int main()
 8  {
 9    // we compute 2^{43,112,609}-1 modulo 10^10 = 10,000,000,000;
10    // this gives us the desired result. Since for all positive
11    // integers a, b, m
12    //    (a+b) mod m = (a mod m + b mod m) mod m , and
13    //    (a*b) mod m = (a mod m * b mod m) mod m,
14    // it follows that we can also compute all intermediate
15    // results modulo 10^10, without making any mistake.
16    // We need long integers, though, since we don't have a
17    // sufficient number of digits otherwise
18
19    ifm::integer m = 100000; // 10^5;
20    m = m * m;               // 10^10;
21
22    // to compute 2^{43,112,609} quickly, we use "repeated squaring":
23    // This works as follows: to compute n^p, we write p as 2q + r,
24    // where r is 0 or 1. Then n^p = (n^2)^q * n^r.
25    unsigned int p = 43112609;
26    ifm::integer n = 2;
27    ifm::integer result = 1;
28    while (p > 0) {
29      unsigned int r = p % 2;
30      if (r > 0)
31        result = result * n % m;  // *= n^r
32      p = p / 2;                  // q
33      n = n * n % m;              // n^2
34    }
35
36    // now subtract 1 and output
37    std::cout << "Last 10 decimal digits: " << result - 1 << "\n";
38
39    return 0;
40  }
```

The ouput is 6697152511.