B.7 A first C++ function

Solution to Exercise 92.

a)
```
// POST: return value is the maximum of i,j and k
```

b)
```
// PRE:  0 not contained in {i,...,j}
// POST: return value is the sum 1/i + 1/(i+1) + ... + 1/j
```

Solution to Exercise 93.

a) If `i` is odd, the execution does not reach a return statement, and the function call expression is invalid. It seems that we want a function that returns `true` if and only if `i` is even. This can be done as follows.

```
bool is_even (int i)
{
  // POST: return value is true if and only if i is even
  return (i % 2 == 0);
}
```

b) If `x` has value 0, `result` will never be set to a defined value. The corresponding function call expression has undefined value. To fix this, we can either make `x != 0.0` a precondition (but then we don't have to check it like the function does it), or we can invent some return value for the case where `x` has value 0. In any case, we don't need the variable `result`. Here are the two variants.

```
double inverse (double x)
{
  // PRE:  x != 0
  // POST: return value is 1/x
  return 1.0 / x;
}

double inverse (double x)
{
  // POST: return value is 1/x for x !=0, and 0 otherwise
  if (x != 0.0)
    return 1.0 / x;
  else
    return 0.0;
}
```

Solution to Exercise 94. The program outputs the 7-th power of the input value i. The computation takes place in the function g that multiplies i with `f(i)` (i^2) and `f(f(i))` (i^4).

Solution to Exercise 95. Here are the three problems.

a) The call of `g(2.0 * x)` in the function body of `f` is not in the scope of the function g, since that function is only declared later through its definition. Consequently, g cannot be used in `f`.

b) In the function g, the modulus operator is used with `double` operands, but for floating point number type operands, there is no modulus operator.

c) The function body of `h` is not in the scope of the variable `result`, since that variable is only declared later. Consequently, `result` cannot be used in `h`.

Solution to Exercise 96.

```
1  // Program: fpsys2.cpp
2  // Provide a graphical representation of floating point numbers
3
4  #include <iostream>
5  #include <cmath>
6  #include <IFM/window>
7
8
9  int main()
10 {
11   // Input parameters of floating point system
12   std::cout << "Draw F(2,p,e_min,e_max).\np =? ";
13   unsigned int p;
14   std::cin >> p;
15   std::cout << "e_min =? ";
16   int emin;
17   std::cin >> emin;
18   std::cout << "e_max =? ";
19   int emax;
20   std::cin >> emax;
21
22   // compute the smallest normalized significand 2^(p-1)
23   const unsigned int smin = (unsigned int)(std::pow (2.0, double(p-1)));
24   // compute the largest normalized significand (2^p)-1
25   const unsigned int smax = 2 * smin - 1;
26   // compute 2^emin
27   const double pemin = std::pow (2.0, double(emin));
28   // compute 2^emax
29   const double pemax = std::pow (2.0, double(emax));
30
31   // For each positive number x of the system draw a circle
32   // with radius x around the window center
33
34   // parameters to scale output
35   const int cx = (ifm::wio.xmax() - ifm::wio.xmin()) / 2;
36   const int cy = (ifm::wio.ymax() - ifm::wio.ymin()) / 2;
37   const double scale = cx / (pemax * smax);
38
39   // zero
```

```
40    ifm::wio << ifm::Point(cx, cy);
41    // loop over all normalized significands
42    for (unsigned int i = smin; i <= smax; ++i)
43      // loop over all exponents
44      for (double m = pemin; m <= pemax; m *= 2)
45        ifm::wio << ifm::Circle(cx, cy, int(m * i * scale));
46
47    ifm::wio.wait_for_mouse_click();
48    return 0;
49  }
```

Solution to Exercise 97. The program still works if $s(x) \geq \sqrt{x}$, so we only have a potential problem if $s(x) < \sqrt{x}$. Because $|s(x) - \sqrt{x}|$ equals $\sqrt{x} - s(x)$ in this case, the relative error bound gives us $\sqrt{x} - s(x) \leq \varepsilon\sqrt{x}$, and this implies

$$s(x) \geq (1 - \varepsilon)\sqrt{x} \geq \frac{1}{2}\sqrt{x}.$$

It follows that

$$2s(x) \geq \sqrt{x},$$

meaning that we can safely use `2*std::sqrt(n)` instead of `std::sqrt(n)` in Program 27. Note that $2s(x)$ is indeed representable as a floating point number again, since we have assumed the system to be binary.

Solution to Exercise 98. The desired number of twin primes is 58980 as the following program shows that implements the approach of a). It turns out that we have to use the fast prime number test from Program 27 in order not to wait too long (we still have to wait pretty long).

```
1   // Program: twinprimes.cpp
2   // Count twin primes in 2,...,10000000
3
4   #include <iostream>
5   #include <cmath>
6
7   // POST: return value is true if and only if n is prime
8   bool is_prime (const unsigned int n)
9   {
10    if (n < 2) return false; // 0 and 1 are not prime
11
12    // Computation: test possible divisors d up to sqrt(n)
13    const unsigned int bound = (unsigned int)(std::sqrt(n));
14    unsigned int d;
15    for (d = 2; d <= bound && n % d != 0; ++d);
16
17    // Output
18    return d > bound;
19  }
20
21  int main ()
22  {
23    // keep primality info for odd i and i+2
24    bool curr = false; // i = 1
```

```
25    bool next = true;  // i = 3
26    int twins = 0;     // number of twins
27    for (int i = 3; i < 9999999; i += 2) {
28      curr = next;          // i
29      next = is_prime(i+2); // i+2
30      if (curr && next) ++twins;
31    }
32    std::cout << "Number of twin primes: " << twins << "\n";
33
34    return 0;
35  }
```

A much faster approach is based on *Eratosthenes's sieve*. We simply compute all prime numbers in the given range, and then use this information to select the twin primes:

```
1   // Program: twinprimes2.cpp
2   // Count twin primes in 2,...,10000000
3
4   #include <iostream>
5   #include <algorithm>
6
7   int main()
8   {
9     // definition and initialization: provides us with
10    // Booleans crossed_out[0],..., crossed_out[9999999]
11    bool crossed_out[10000000];
12    std::fill (crossed_out, crossed_out + 10000000, false);
13
14    // computation of all prime numbers in the range
15    for (unsigned int i = 2; i < 10000000; ++i)
16      if (!crossed_out[i])
17        // cross out all proper multiples of i
18        for (unsigned int m = 2*i; m < 10000000; m += i)
19          crossed_out[m] = true;
20
21    // now count twin primes: (i-2, i) = (3,5) is first pair
22    unsigned int twins = 0;
23    for (unsigned int i = 5; i < 10000000; i+=2)
24      if (!crossed_out[i-2] && !crossed_out[i])
25        ++twins;
26
27    // output
28    std::cout << "Number of twin primes: " << twins << "\n";
29
30    return 0;
31  }
```

This shows that the subdivision of the task into subtasks according to a) is not appropriate in this case.

Solution to Exercise 99.

```
double pow (double b, int e)
{
  // PRE:  e >= 0 || b != 0.0
  // POST: return value is b^e
  double result = 1.0;
  if (e < 0) {
```

```
    // b^e = (1/b)^(-e)
    b = 1.0/b;
    e = -e;
  }
  // maintain bpow = b^(2^i), initialized for i=0
  for (double bpow = b; e != 0; e /= 2) {
    if (e % 2 == 1) result *= bpow;
    bpow *= bpow; // square bpow to get b^(2^(i+1))
  }
  return result;
}
```

Solution to Exercise 100.

```
#include<iostream>

// PRE:  i_ptr and j_ptr point to existing objects
// POST: the values of these two objects are swapped
void swap (int* i_ptr, int* j_ptr)
{
  const int h = *i_ptr;
  *i_ptr = *j_ptr;
  *j_ptr = 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 101.

```
// Prog: unique. C
// implemements and tests a function that checks whether every element
// in a sorted sequence is unique

#include<iostream>
#include<cassert>

// PRE:  [first, last) is a valid range and describes a sequence
//       of elements that are sorted in nondecreasing order
// POST: the return value is true if and only if no element
//       occurs twice in the sequence
```

```
12  bool unique (const int* first, const int* last)
13  {
14    if (first == last)
15      return true;
16    --last;
17    for (const int* p = first; p < last;) {
18      const int curr = *p;
19      const int next = *++p;
20      assert (curr <= next);
21      if (curr == next)
22        return false;
23    }
24    return true;
25  }
26
27  int main()
28  {
29    int a[5] = {1,2,2,3,4};
30    int b[5] = {5,6,7,8,9};
31    std::cout << unique (a, a+5) << "\n"
32              << unique (b, b+5) << "\n";
33
34    return 0;
35  }
```

Solution to Exercise 102. We simply take the sorting loop out of `sort_array.cpp` and put it into a function. At the same time, we move from iteration by index to iteration by pointers. A less elegant but also valid solution is to keep the original code with iteration by index, after computing `n` as `last-first`.

```
1   // Program: sort_array2.cpp
2   // read a sequence of n numbers into an array,
3   // sort them using a function, and output the
4   // sorted sequence
5   #include <iostream>
6   #include <algorithm>
7
8   // PRE: [first, last) is a valid range
9   // POST: the elements *p, p in [first, last) are
10  //       in ascending order
11  void sort (int* first, int* last)
12  {
13    // sort array: in round p=first,...,last-1 we find
14    // the smallest element in the range described by
15    // [p, last) and interchange it with *p
16    for (int* p = first; p != last; ++p) {
17      // find minimum in nonempty range described by [p, last)
18      int* p_min = p; // pointer to current minimum
19      int* q = p;     // pointer to current element
20      while (++q != last)
21        if (*q < *p_min) p_min = q;
22      // interchange *p with *p_min
23      std::iter_swap (p, p_min);
24    }
25  }
26
27  int main()
28  {
29    // input of n
30    unsigned int n;
31    std::cin >> n;
```

```
32
33    // dynamically allocate array
34    int* const a = new int[n];
35
36    // read into the array
37    for (int i=0; i<n; ++i) std::cin >> a[i];
38
39    // sort
40    sort (a, a+n);
41
42    // output sorted sequence
43    for (int i=0; i<n; ++i) std::cout << a[i] << " ";
44    std::cout << "\n";
45
46    // delete array
47    delete[] a;
48
49    return 0;
50
51  }
```

Solution to Exercise 103. The postcondition is

```
// POST: The range [b,e) is copied in reverse order into the
// range [o, o+(e-b))
```

In b), the first call is invalid, since `[a+5,a+10)` is not a valid range. The second call is ok, but the third one is again invalid, since the ranges `[a,a+3)` and `[a+2,a+5)` overlap (in one element). For c), we observe that the elements described by the source range `[b,e)` are not modified. Thus, the pointers `b` and `e` should both have type `const int*`.

Solution to Exercise 104. The 5 values of `b[0]` up to `b[4]` after the function call are $(1,4,6,4,1)$. Incidentally, this is the fifth row of *Pascal's triangle*

$$\begin{array}{ccccccccccc}
 & & & & & 1 & & & & & \\
 & & & & 1 & & 1 & & & & \\
 & & & 1 & & 2 & & 1 & & & \\
 & & 1 & & 3 & & 3 & & 1 & & \\
 & 1 & & 4 & & 6 & & 4 & & 1 & \\
1 & & 5 & & 10 & & 10 & & 5 & & 1 \\
 & & & & & \vdots & & & & &
\end{array}$$

where every entry is the sum of the two entries directly above it. And it is not hard to prove that this generalizes. If `e - b` has value n, the n values `b[0]` up to `b[n-1]` form the n-th row of Pascal's triangle. We prove this by induction on n, where the case $n = 1$ is easy (`b[0]==1`). Now let $n > 1$ and assume that the statement holds for $n-1$. In going from $n-1$ to n, there is one more outer loop iteration (with $i = n$) that first sets `b[n-1]` to 1 (as desired). The values `b[n-2]`, `b[n-3]`, `b[1]` that by hypothesis form the $(n-1)$-st row of Pascal's triangle (without the first element) are updated as follows: to every such element, we add the one directly to the left of it. By definition of Pascal's triangle, this yields the correct elements in the n-th row.

We would like to remark that the k-th entry in the n-th row of Pascal's triangle is the binomial coefficient $\binom{n-1}{k-1}$; see also Exercise 115.

Solution to Exercise 105. Given the input date, the main task is to count the number of days that have passed since January 1, 1900. Once we have that information, we can take the number modulo 7, and this determines the weekday.

Here is a list of the major subtasks (your list might be different).

1. find out whether the input date is legal;
2. count the number of days in all *years* preceding the input date;
3. count the number of days in all *months* preceding the input date (this has 2 as a subtask);
4. count the number of days preceding the input date (this has 3 as a subtask);
5. given this count, output the weekday.

Smaller subtasks needed by the above are the following.

1. find out whether a given year is a leap year;
2. find out how many days a given month has in a given year;
3. compute the weekday as a number in $\{0, \ldots, 6\}$;
4. transform the number of a weekday into its name.

Here is the program resulting from this subdivision into subtasks.

```
1  // Prog: perpetual_calendar
2  // compute the weekday for any given date >= 01.01.1900 (Monday)
3
4  #include<iostream>
5  #include<cassert>
6
7  // PRE:  year >= 1900
8  // POST: return value is true iff year is a leap year
9  // ------------------------------------------------
10 bool is_leap_year (const unsigned int year)
11 {
12   assert (year >= 1900);
13   return year % 4 == 0 && (year % 100 != 0 || year % 400 == 0);
14 }
15
16 // PRE:  year >= 1900, 1 <= month <= 12
17 // POST: return value is the number of days in month.year
18 // ------------------------------------------------------
19 unsigned int days_in_month (const unsigned int month,
20                             const unsigned int year)
21 {
22   assert (year >= 1900);
23   assert (1 <= month && month <= 12);
24   unsigned int days[12] =
```

```
25      {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
26    return days[month-1] + (month==2 && is_leap_year (year));
27  }
28
29  // POST: return value is true iff day.month.year is
30  //       an existing date  >= 01.01.1900
31  // ------------------------------------------------
32  bool is_date (const unsigned int day, const unsigned int month,
33                const unsigned int year)
34  {
35    return
36      year >= 1900 && 1 <= month && month <= 12 &&
37      1 <= day && day <= days_in_month (month, year);
38  }
39
40
41  // PRE:  year >= 1900
42  // POST: return value is the number of days in years [1900, ..., year)
43  // ---------------------------------------------------------------------
44  unsigned int days_up_to_year (const unsigned int year)
45  {
46    assert (year >= 1900);
47    unsigned int d = 0;
48    // the following could be done more efficiently, but why?
49    for (unsigned int y = 1900; y < year; ++y)
50        d += 365 + is_leap_year (y);
51    return d;
52  }
53
54  // PRE:  year >= 1900, 1 <= month <= 12
55  // POST: return value is the number of days in
56  //       months [01.1900, ..., month.year)
57  // ----------------------------------------
58  unsigned int days_up_to_month (const unsigned int month,
59                                 const unsigned int year)
60  {
61    assert (year >= 1900);
62    assert (1 <= month && month <= 12);
63    unsigned int days = days_up_to_year (year); // before year
64    for (unsigned int m = 1; m < month; ++m)    // before month
65      days += days_in_month (m, year);
66    return days;
67  }
68
69  // PRE:  is_date (day, month, year)
70  // POST: return value is the number of days in
71  //       [01.01.1900, ..., day.month.year)
72  // -------------------------------------
73  unsigned int days_up_to (const unsigned int day,
74                           const unsigned int month,
75                           const unsigned int year)
76  {
77    assert (is_date (day, month, year));
78    return days_up_to_month (month, year) + day - 1;
79  }
80
81  // PRE:  is_date (day, month, year)
82  // POST: return value is weekday (0 = Monday,...,6 = Sunday)
83  // ---------------------------------------------------------
84  unsigned int weekday (const unsigned int day,
85                        const unsigned int month,
86                        const unsigned int year)
87  {
88    assert (is_date (day, month, year));
89    return days_up_to (day, month, year) % 7;
```

```
90  }
91
92  // PRE:  weekday < 7
93  // POST: writes the name of the weekda to standard output
94  //       (0 = Monday,...,6 = Sunday)
95  // ---------------------------------
96  void print_weekday (const unsigned int weekday) {
97    assert (weekday < 7);
98    if      (weekday == 0) std::cout << "Monday";
99    else if (weekday == 1) std::cout << "Tuesday";
100   else if (weekday == 2) std::cout << "Wednesday";
101   else if (weekday == 3) std::cout << "Thursday";
102   else if (weekday == 4) std::cout << "Friday";
103   else if (weekday == 5) std::cout << "Saturday";
104   else                   std::cout << "Sunday";
105   std::cout << "\n";
106 }
107
108 int main()
109 {
110   // input date
111   std::cout << "Compute weekday of date (day/month/year) for\n";
112   std::cout << "day =? ";
113   unsigned int day; std::cin >> day;
114   std::cout << "month =? ";
115   unsigned int month; std::cin >> month;
116   std::cout << "year =? ";
117   unsigned int year; std::cin >> year;
118
119   // check date
120   if (!is_date (day, month, year)) {
121     std::cout << "Illegal date.\n";
122     return 1;
123   }
124
125   // output weekday
126   print_weekday (weekday (day, month, year));
127
128   return 0;
129 }
```

Solution to Exercise 106. Here we show how it is done under Unix-type platforms and the g++ compiler. Let's assume that your home directory is `/home/myhome/`. Under this directory, you now create a subdirectory named `libifm`, with two subdirectories `include` and `lib`. In the subdirectory `include`, you create another subdirectory `IFM`.

Now you copy `math.h` into the `include/IFM` subdirectory and `math.cpp` into the `lib` subdirectory.

(i) Build the object code. For this, go to the `lib` subdirectory and tell the compiler to generate the object code file `math.o`. For this, you type

```
g++ -I/home/myhome/libifm/include -c math.cpp
```

The directory after the `-I` is the one where the compiler will look for your include files. The path `IFM/math.h` in `math.cpp` is relative to that include directory. You can provide several include directories through several `-I`'s.

(ii) Build the library from the o-files in the `lib` subdirectory. For this, you type

```
ar r libmath.a *.o
```

This tells the archive program `ar` to put all object files found in the current directory into a single library called `libmath.a`.

(iii) Build the executable from whatever directory you want, by typing

```
g++ -I/home/myhome/libifm/include -L/home/myhome/libifm/lib
    callpow4.cpp -lmath -o callpow4
```

This tells g++ to compile (and link) the program `callpow4.cpp`, using the library `libmath.a` found in the directory that is specified after `-L`.

Solution to Exercise 107.

By the C++ standard, converting a floating point number to `int` cuts off the fractional part. Then we compute the error between the original number and its truncation. Under the IEEE standard 754, this difference has one significant digit less than and is therefore exactly representable as a `double` value, *unless* the input value is negative and has largest possible exponent. But in this case, the truncation is also out of range, and we don't care. Now we can exactly test whether the error is at most 0.5 (in which case we return the truncation), or more (in which case we return the next integer, going away from zero).

```
1  #include <iostream>
2
3  // POST: return value is the integer nearest to x; if there are
4  //       two nearest integers, the one closer to 0 is chosen
5  int round (const double x)
6  {
7    const int trunc = int(x); // rounds towards 0 by standard
8    const double error = x - trunc; // note: result is exact!
9    if (error > 0.5)
10     // x was positive, and trunc + 1 is nearer
11     return trunc + 1;
12   if (error < -0.5)
13     // x was negative, and trunc - 1 is nearer
14     return trunc - 1;
15   // |error| <= 0.5, trunc is closest integer
16   return trunc;
17 }
18
19 int main ()
20 {
21   for (double d = -2; d <= 2; d += 0.25)
22     std::cout << "closest integer to " << d << " is "
23               << round (d) << "\n";
24
25   return 0;
26 }
```

As far as the integration into the library is concerned, please follow the procedure in the solution to Exercise 106 above.

Solution to Exercise 108. We apply a slightly tuned brute-force solution (but maybe you found something better). We in fact compute all possible powers, and for each one compute the cross sum. We use the type `ifm::integer` in order to have sufficient precision. The only slight improvement is that we compute the cross sum not by the straightforward method (going through the number digit by digit), but by breaking the number up into larger chunks through using a divisor larger than 10. On the platform of the authors, the divisor in the program below gives the fastest program. Its output is

```
Best a = 99
Best b = 95
```

The program below is too slow for $a, b < 1000$, so we don't know the answer for this case.

```
1  // Prog: power_cross_sums.cpp
2  // find the values a and b smaller than 100 such that
3  // a^b has maximum cross sum
4
5  #include <iostream>
6  #include <IFM/integer.h>
7
8  // POST: returns sum of decimal digits of n
9  ifm::integer cross_sum (ifm::integer n)
10 {
11   // the straightforward method
12   ifm::integer cross = 0;
13   for (; n > 0; n /= 10) {
14     cross += n % 10;
15   }
16   return cross;
17 }
18
19 // POST: returns sum of decimal digits of n
20 ifm::integer cross_sum_big (ifm::integer n)
21 {
22   // a faster method; use a larger divisor
23   // d to reduce n to n /d, and apply the
24   // straightforward method to n % d
25   const ifm::integer d = 10000000;
26   ifm::integer cross = 0;
27   while (n > 0) {
28     ifm::integer k = n / d;
29     cross += cross_sum (n - k * d); // n % d
30     n = k;
31   }
32   return cross;
33 }
34
35 int main()
36 {
37   const int range = 100;
38
39   ifm::integer max_cross_sum = 0;
40   unsigned int best_a = 0;
41   unsigned int best_b = 0;
42   for (int a=1; a<range; ++a) {
```

```
43      ifm::integer power = 1; // a^0
44      for (int b=1; b<range; ++b) {
45        // update to a^b
46        power *= a;
47        // count sum of digits
48        ifm::integer s = cross_sum_big (power);
49        if (s >= max_cross_sum) {
50          max_cross_sum = s;
51          best_a = a;
52          best_b = b;
53        }
54      }
55    }
56
57    // output
58    std::cout << "Best a = " << best_a << "\n";
59    std::cout << "Best b = " << best_b << "\n";
60  }
```

Solution to Exercise 109.

a) The idea is simple: we need to keep track of the parity (even or odd) of the number of zeros and ones processed so far. There are four possible parity combinations, and we have one state for each of them (ee, oe, eo, oo, where the first letter is for the parity of the zeros, the second for the parity of ones). When we are in state eo, for example, and we process the symbol 1, we move to state ee, because the parity of the number of ones changes. We accept if and only if we are in state ee which is also the starting state.

b) While we are processing w, we keep track of the value $w' \bmod 5$, where w' is the prefix of w processed so far. Initially, w' is empty and $w' \bmod = 0$. When we process the next symbol, it may be 0 and we have $w' := 2w'$, or it is 1 in which case we get $w' := 2w' + 1$. In all cases, we can say how $w' \bmod 5$ changes; here is the table.

$w' \bmod 5$	$2w' \bmod 5$	$(2w'+1) \bmod 5$
0	0	1
1	2	3
2	4	0
3	1	2
4	3	4

We therefore need states $0, 1, 2, 3, 4$ corresponding to $w' \bmod 5$, connected by transitions as given in the above table, and we accept if and only if we are in state 0 in the end.

c) This language cannot be the language of any DFA. The intuitive reason is that an automaton can only "count up to some finite number", because it has only finitely many states. But in order to be able to decide whether a word contains more zeros than ones, it would have to count zeros and ones in arbitrarily long words.

But we van also prove formally that there cannot be a DFA for this language L. Assume for a contradiction that there is a DFA whose language is L, and assume that it has n states. The DFA surely accepts the word

$$\underbrace{0....0}_{n+1}\underbrace{1....1}_{n},$$

since it has more zeros than ones. Let s_i be the state the DFA is in after processing the i-th one, $i = 1, \ldots, n$. s_0 is the state before processing the first one. In total, these are $n+1$ states, and since there are only n distinct states, at least one state must occur twice in the sequence $s_0, s_1, \ldots, s_n$. So let $0 \leq j < k \leq n$ be indices such that $s_j = s_k$. This means that if the automaton is in state s_j and then processes $k-j$ ones, it gets back to state $s_j = s_k$: a loop. But then it would also get back to state j after $2(k-j)$ ones. Consequently, the DFA also accepts the word

$$\underbrace{0....0}_{n+1}\underbrace{1....1}_{n+k-j}$$

resulting from w by inserting $k-j$ additional ones after the k-th one. But this is a contradiction, since the latter word does not contain more zeros than ones.

This proof is actually an incarnation of the *pumping lemma*, a tool to prove that a language cannot be the language of a DFA. The class of languages L for which a DFA with language L exists is well-understood; it is called the class of *regular* languages.

d) Here we use the finite counting ability of DFA: we keep track of the longest run of ones that ends in the current symbol. Runs longer than two don't need to be counted, since we already know then that we will reject the word.

Solution to Exercise 110. The code is subdivided into several functions. The most important function `deducible` tries to make one Sherlock-Holmes-type deduction for given r, c and n. It is repeatedly called by the function `fill_cell` for all values of r, c, and n; if a deduction is found, `fill_cell` accordingly updates the `board`, a 3-dimensional array that maintains the information which numbers are candidates for which cells.

The function `deduce` tries the Sherlock-Holmes-type deductions in turn, where it distinguishes between deduction from row, column, or box in case 2.

```
1  // Prog: sudoku.cpp
2  // solves sudokus according to simple deduction heuristic (may fail)
3  #include<cassert>
4  #include<iostream>
5
6  // Here is the algorithm: given a partial filling of the 81 cells,
7  // we try to find a row r, a column c, and a number n such that n is
8  // the unique candidate to be filled into the empty cell (r,c) in row
9  // r and column c. There are two situations in which this is easy:
10 //  1. all numbers distinct from n already appear in the row, column,
```

```
11  //     or 3x3 box containing the cell (r,c)
12  //  2. we already know that n cannot appear in the other cells of the
13  //     row, column, or box containing (r,c)
14  // To check this, we maintain for every triple (r,c,n) the information
15  // whether n is still a candidate for cell (r,c). Whenever a cell is filled,
16  // we remove the filled number from the candidate list of all other cells
17  // in the same row, column, or box.
18
19  // the grid: a 3x3x3 array of boolean values, one for each triple (r,c,n)
20  //   grid[r][c][0] == true iff cell (r,c) not filled yet
21  //   grid[r][c][n] == true for n>0 iff n is still a candidate for cell (r,c)
22  bool grid[9][9][10];
23
24  // PRE:  cell (r,c) is empty, and n>0 is a candidate for (r,c)
25  // POST: puts n into (r,c); removes n as candidate from all other cells
26  //       of the row, the column, and the box of (r,c); removes all
27  //       numbers distinct from n as candidates from (r,c)
28  void update_grid (const unsigned int r, const unsigned int c,
29                    const unsigned int n)
30  {
31    // assert precondition
32    assert (grid[r][c][0] && n > 0 && grid[r][c][n]);
33    // go through row of (r,c)
34    for (unsigned int j=0; j<9; ++j)
35      grid[r][j][n] = false;
36    // go through column of (r,c)
37    for (unsigned int i=0; i<9; ++i)
38      grid[i][c][n] = false;
39    // go through box of (r,c)
40    const unsigned int lr = r-r%3; // (lr, lc) is the lower left cell of
41    const unsigned int lc = c-c%3; // this box
42    for (unsigned int i=lr; i<lr+3; ++i)
43      for (unsigned int j=lc; j<lc+3; ++j)
44        grid[i][j][n] = false;
45    // go through numbers
46    for (unsigned int m=0; m<10; ++m)
47      grid[r][c][m] = false;
48    // fill cell
49    grid[r][c][n] = true;
50  }
51
52  // POST: returns n > 0 if n is the unique candidate for cell (r,c) and
53  //       0 otherwise
54  int unique_number (const unsigned int r, const unsigned int c)
55  {
56    int n = 0;
57    bool found = false;
58    for (unsigned int m = 1; m<10; ++m)
59      if (grid[r][c][m]) {
60        if (found) return 0; // we've already seen a candidate
61        n = m;               // this is the first candidate
62        found = true;
63      }
64    return n;
65  }
66
67  // PRE:  cell (r,c) is empty, and n>0 is a candidate for (r,c)
68  // POST: returns true if and only if n is not a candidate for any
69  //       cell distinct from (r,c) within the same row
70  bool deducible_from_row (const unsigned int r, const unsigned int c,
71                           const unsigned int n)
72  {
73    // assert preconditions
74    assert (grid[r][c][0] && n > 0 && grid[r][c][n]);
75    // go through the columns
```

```
76    for (int j=0; j<9; ++j)
77      if (j != c && grid[r][j][n]) return false; // candidate somewhere else
78    return true;
79  }
80
81  // PRE:  cell (r,c) is empty, and n>0 is a candidate for (r,c)
82  // POST: returns true if and only if n is not a candidate for any
83  //       cell distinct from (r,c) within the same column
84  bool deducible_from_column (const unsigned int r, const unsigned int c,
85                              const unsigned int n)
86  {
87    // assert preconditions
88    assert (grid[r][c][0] && n > 0 && grid[r][c][n]);
89    // go through the rows
90    for (int i=0; i<9; ++i)
91      if (i != r && grid[i][c][n]) return false; // candidate somewhere else
92    return true;
93  }
94
95  // PRE:  cell (r,c) is empty, and n>0 is a candidate for (r,c)
96  // POST: returns true if and only if n is not a candidate for any
97  //       cell distinct from (r,c) within the same box
98  bool deducible_from_box (const unsigned int r, const unsigned int c,
99                           const unsigned int n)
100 {
101   // assert preconditions
102   assert (grid[r][c][0] && n > 0 && grid[r][c][n]);
103   const unsigned int lr = r-r%3; // (lr, lc) is the lower left cell of
104   const unsigned int lc = c-c%3; // this box
105   // go through the box
106   for (int i=lr; i<lr+3; ++i)
107     for (int j=lc; j<lc+3; ++j)
108       if ( (i != r || j != c) && grid[i][j][n])
109         return false; // candidate somewhere else
110   return true;
111 }
112
113 // POST: returns true iff (r,c) is empty, n>0 is a candidate for (r,c), and
114 //         - n is the unique candidate for the cell (r,c), or
115 //         - n is deducible as the unique candidate from the row,
116 //           column, or box of (r,c)
117 bool deducible (const unsigned int r, const unsigned int c,
118                 const unsigned int n)
119 {
120   if (grid[r][c][0] && n > 0 && grid[r][c][n])
121     return
122       (n == unique_number (r,c))      ||  // unique candidate for (r,c) ?
123       deducible_from_row (r, c, n)    ||
124       deducible_from_column (r, c, n) ||
125       deducible_from_box (r, c, n);
126   return false;
127 }
128
129 // POST: returns true if and only if there is an empty cell (r,c) and a
130 //       number n>0 such that n can be deduced to be the number to be
131 //       put into (r,c); if the return value is true, the grid is updated
132 //       by filling an empty cell with an accordingly deduced number
133 bool fill_cell()
134 {
135   for (unsigned int r=0; r<9; ++r)
136     for (unsigned int c=0; c<9; ++c)
137       for (unsigned int n=1; n<10; ++n)
138         if (deducible (r, c, n)) {
139           update_grid (r, c, n);
140           return true;
```

```
141           }
142     return false;
143   }
144
145   int main()
146   {
147     // set up empty grid
148     for (int r=0; r<9; ++r)
149       for (int c=0; c<9; ++c) {
150         for (int n=0; n<10; ++n)
151           grid[r][c][n] = true;
152       }
153
154     // input: 9 x 9 numbers in {0,..,9} rowwise (0 means no number)
155     unsigned int filled_cells = 0; // total number of filled cells
156     for (int r=0; r<9; ++r)
157       for (int c=0; c<9; ++c) {
158         int n;
159         std::cin >> n;
160         if (n > 0) {
161           update_grid (r, c, n);
162           ++filled_cells;
163         }
164       }
165
166     // main loop
167     while (fill_cell()) ++filled_cells;
168
169     // sudoku is solved if all cells could be filled
170     if (filled_cells == 81)
171       std::cout << "Sudoku is solved:\n";
172     else
173       std::cout << "Could only fill " << filled_cells << " cells:\n";
174     // output solution in 3 x 3 blocks (0 means no number deduced);
175     for (unsigned int lr=0; lr<9; lr+=3) {
176       for (unsigned int r=lr; r<lr+3; ++r) {
177         for (unsigned int lc=0; lc<9; lc+=3) {
178           for (unsigned int c=lc; c<lc+3; ++c)
179             std::cout << unique_number (r, c) << " "; // blank after number
180           std::cout << " "; // extra blank after every three columns
181         }
182         std::cout << "\n"; // extra line break after every third row
183       }
184       std::cout << "\n"; // line break after row
185     }
186   }
```

Solution to Exercise 111.

Fuzzy Products. Let's go step by step. Mr. Product says "I don't know the numbers a and b." This means that there must be more than one set $\{a, b\}$ such that $p = a \cdot b$. Let us call such a number p a *fuzzy product*. For example, $p = 12$ is a fuzzy product, because $12 = 2 \cdot 6 = 3 \cdot 4$. The number $p = 6$ is not a fuzzy product, since the only possible set is $\{2, 3\}$ (recall that $\{1, 6\}$ is not an option due to $1 < a, b$.

So, after the first statement of Mr. Product, we and Mr. Sum know that p must be fuzzy product.

Fuzzy Sums. Mr. Sum then says: "I knew that you don't know them." We can conclude that s must be a number, such that, for *every* way of writing it as a sum of two numbers a and b, the product $a \cdot b$ is a fuzzy product. Otherwise Mr. Sum would not have been able to be sure about his statement. Let us call such a number s a *fuzzy sum*. For example, $s = 11$ is a fuzzy sum, because

$$\begin{array}{rcl} 11 = 2+9 & \text{and} & 2 \cdot 9 = 18 = 3 \cdot 6 \\ 11 = 3+8 & \text{and} & 3 \cdot 8 = 24 = 4 \cdot 6 \\ 11 = 4+7 & \text{and} & 4 \cdot 7 = 28 = 2 \cdot 14 \\ 11 = 5+6 & \text{and} & 5 \cdot 6 = 30 = 3 \cdot 10 \end{array}$$

So, after the first statement of Mr. Sum, we and Mr. Product know that s must be fuzzy sum.

Rare Products. In his mind, Mr. Product quickly computes all fuzzy sums between 4 and 198 (these are the candidates), and finds that there are only 10 of them: They are $11, 17, 23, 27, 29, 35, 37, 41, 47, 53$, and $s = a+b$ must be one of them. Now, Mr. Product, knowing p, is trying to solve— for each of the 10 candidates s—the equation system

$$\begin{array}{rcl} s & = & a+b \\ p & = & ab \end{array}$$

in order to find the two unknowns a and b. Since his next statement is "Ah... but now I know them.", we can conclude that the system has a valid solution for *exactly one* of the 10 fuzzy sums s. Therefore, p must have another special property: exactly one of the sets $\{a, b\}$ with $a \cdot b = p$ has a fuzzy sum $a + b$. We call such a p a *rare product*.

So, after the second statement of Mr. Product, we and Mr. Sum know that p must be a rare product.

Rare Sums. In his mind, Mr. Sum, knowing s, quickly computes all sets $\{a, b\}$ such that $s = a + b$ and $a \cdot b$ is a rare product. After this, he says "Then I know them too, now." We conclude that *exactly one* set $\{a, b\}$ such that $s = a + b$ has a rare product $a \cdot b$. We call such an s a *rare sum*. Mr. Sum now knows $\{a, b\}$, but we still have some work to do.

After the second statement of Mr. Sum, we know that s is both a fuzzy and a rare sum. And in order for the puzzle to be solvable for us, we know that exactly one s in the range $3 < s < 199$ must have this property.

Putting it Together. Now we write a program to search for the supposedly unique number $3 < s < 199$ that is both a fuzzy and a rare sum. And from the rare sum property of s,

we can then uniquely deduce $\{a, b\}$. It turns out that

$$\begin{aligned} s &= 17, \\ p &= 52, \\ \{a, b\} &= \{4, 13\}. \end{aligned}$$

is indeed the unique solution.

```
1  // Prog: mr_prod_sum.cpp
2  // solves the puzzle of Mr. Sum and Mr. Product
3  #include <iostream>
4  #include <cassert>
5
6  // PRE:  1 < a < 100, s > 3
7  // POST: returns true iff 1 < a <= b := s-a < 100
8  bool legal_sum (int s, int a, int& b) {
9    assert (s > 3);
10   assert ((1 < a) && (a < 100));
11   b = s - a;
12   return ((a <= b) && (b < 100));
13 }
14
15 // PRE:  1 < a < 100, p > 3
16 // POST: returns true iff a divides p, and 1 < a <= b := p/a < 100
17 bool legal_product (int p, int a, int& b) {
18   assert (p > 3);
19   assert ((1 < a) && (a < 100));
20   b = p / a;
21   return ((p % a == 0) && (a <= b) && (b < 100));
22 }
23
24 // PRE:  p > 3
25 // POST: returns true iff p can be written in more
26 //       than one way as the product of two numbers
27 //       between 2 and 99
28 bool fuzzy_product(int p) {
29   assert (p > 3);
30   int count = 0;
31   int b;
32   for (int a = 2; a < 100; ++a) {
33     if (legal_product (p, a, b))
34       ++count;
35   }
36   return count > 1;
37 }
38
39 // PRE:  s > 3
40 // POST: Returns true iff for all a,b such that
41 //       s = a+b, the product a*b is a fuzzy product.
42 bool fuzzy_sum(int s) {
43   assert (s > 3);
44   int b;
45   for (int a = 2; a < 100; ++a) {
46     if (legal_sum (s, a, b) && !fuzzy_product( a * b )) {
47       return false;
48     }
49   }
50   return true;
51 }
52
53 // PRE:  p > 3
54 // POST: returns true iff exactly one of the sets {a,b}
```

```
55  //       with p = a*b has a fuzzy sum a+b.
56  bool rare_product(int p) {
57    assert (p > 3);
58    int count = 0;
59    int b;
60    for (int a = 2; a < 100; ++a) {
61      if (legal_product (p, a, b) && fuzzy_sum (a + b))
62        ++count;
63    }
64    return count == 1;
65  }
66
67
68  // PRE:  s > 3
69  // POST: returns true iff exactly one of the pairs (a,b)
70  //       with a+b=s has a rare product a*b.
71  //       If the return value is true, then the corresponding
72  //       pair {a,b} can be retrieved from the parameters.
73  bool rare_sum(int s, int& a, int& b) {
74    assert (s > 3);
75    int count = 0;
76    int bb;
77    for (int aa = 2; aa < 100; ++aa) {
78      if (legal_sum (s, aa, bb) && rare_product( aa * bb )) {
79        a = aa;
80        b = bb;
81        ++count;
82      }
83    }
84    return count == 1;
85  }
86
87  int main() {
88    std::cout << "Compute a and b in the range [2..99]...\n";
89    int a;
90    int b;
91
92    for (int s = 4; s < 199; ++s) {
93      //std::cout << "Checking sum: " << i << std::endl;
94      if (fuzzy_sum(s) && rare_sum(s, a, b)) {
95        std::cout << "Solution:\n";
96        std::cout << "s= " << s << "\n";
97        std::cout << "p= " << a*b << "\n";
98        std::cout << "a= " << a << "\n";
99        std::cout << "b= " << b << "\n";
100     }
101   }
102
103   return 0;
104 }
```