Solution to Exercise 79.

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 80.

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 81. 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 82. 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 83.

```
1  // Program: fpsys2.C
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   unsigned int smin = (unsigned int)(std::pow (2.0, double(p-1)));
24   // compute the largest normalized significand (2^p)-1
25   unsigned int smax = 2 * smin - 1;
26   // compute 2^emin
27   double pemin = std::pow (2.0, double(emin));
28   // compute 2^emax
29   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   int cx = (ifm::wio.xmax() - ifm::wio.xmin()) / 2;
36   int cy = (ifm::wio.ymax() - ifm::wio.ymin()) / 2;
37   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 84. 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 28. 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 85. 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 28 in order not to wait too long (we still have to wait pretty long).

```
 1  // Program: twinprimes.C
 2  // Count twin primes in 2,...,10000000
 3
 4  #include <iostream>
 5  #include <cmath>
 6
 7  bool is_prime (unsigned int n)
 8  {
 9    // POST: return value is true if and only if n is prime
10    if (n < 2) return false; // 0 and 1 are not prime
11
12    // Computation: test possible divisors d up to sqrt(n)
13    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.C
 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 86.

```
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 87.

```
1  #include<iostream>
2
3  // PRE:  i_ptr and j_ptr point to existing objects
4  // POST: the values of these two objects are swapped
5  void swap (int* i_ptr, int* j_ptr)
6  {
7    int h = *i_ptr;
8    *i_ptr = *j_ptr;
9    *j_ptr = h;
10 }
11
12 int main() {
13   // input
14   std::cout << "i =? ";
15   int i; std::cin >> i;
16
17   std::cout << "j =? ";
18   int j; std::cin >> j;
19
20   // function call
21   swap(&i, &j);
22
23   // output
24   std::cout << "Values after swapping: i = " << i
25             << ", j = " << j << ".\n";
26
27   return 0;
28 }
```

Solution to Exercise 88. We simply take the sorting loop out of `sort_array.C` 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.C
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* 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 89. 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 (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 (unsigned int month, unsigned int year)
20 {
21   assert (year >= 1900);
22   assert (1 <= month && month <= 12);
23   static unsigned int days[12] =
24     {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
25   return days[month-1] + (month==2 && is_leap_year (year));
26 }
27
28 // POST: return value is true iff day.month.year is
29 //       an existing date  >= 01.01.1900
30 // ---------------------------------------------
31 bool is_date (unsigned int day, unsigned int month, unsigned int year)
32 {
33   return
34     year >= 1900 && 1 <= month && month <= 12 &&
35     1 <= day && day <= days_in_month (month, year);
36 }
37
38
39 // PRE:  year >= 1900
40 // POST: return value is the number of days in years [1900, ..., year)
41 // --------------------------------------------------------------------
42 unsigned int days_up_to_year (unsigned int year)
43 {
44   assert (year >= 1900);
45   unsigned int d = 0;
46   // the following could be done more efficiently, but why?
47   for (unsigned int y = 1900; y < year; ++y)
48       d += 365 + is_leap_year (y);
49   return d;
50 }
51
52 // PRE:  year >= 1900, 1 <= month <= 12
53 // POST: return value is the number of days in
54 //       months [01.1900, ..., month.year)
55 // ------------------------------------------
```

```
56  unsigned int days_up_to_month (unsigned int month, unsigned int year)
57  {
58    assert (year >= 1900);
59    assert (1 <= month && month <= 12);
60    unsigned int days = days_up_to_year (year); // before year
61    for (unsigned int m = 1; m < month; ++m)    // before month
62      days += days_in_month (m, year);
63    return days;
64  }
65
66  // PRE:  is_date (day, month, year)
67  // POST: return value is the number of days in
68  //       [01.01.1900, ..., day.month.year)
69  // ---------------------------------------
70  unsigned int days_up_to (unsigned int day, unsigned int month,
71                           unsigned int year)
72  {
73    assert (is_date (day, month, year));
74    return days_up_to_month (month, year) + day - 1;
75  }
76
77  // PRE:  is_date (day, month, year)
78  // POST: return value is weekday (0 = Monday,...,6 = Sunday)
79  // ---------------------------------------------------------
80  unsigned int weekday (unsigned int day,
81                        unsigned int month,
82                        unsigned int year)
83  {
84    assert (is_date (day, month, year));
85    return days_up_to (day, month, year) % 7;
86  }
87
88  // PRE:  weekday < 7
89  // POST: writes the name of the weekda to standard output
90  //       (0 = Monday,...,6 = Sunday)
91  // --------------------------------
92  void print_weekday (unsigned int weekday) {
93    assert (weekday < 7);
94    if      (weekday == 0) std::cout << "Monday";
95    else if (weekday == 1) std::cout << "Tuesday";
96    else if (weekday == 2) std::cout << "Wednesday";
97    else if (weekday == 3) std::cout << "Thursday";
98    else if (weekday == 4) std::cout << "Friday";
99    else if (weekday == 5) std::cout << "Saturday";
100   else                   std::cout << "Sunday";
101   std::cout << "\n";
102 }
103
104 int main()
105 {
106   // input date
107   std::cout << "Compute weekday of date (day/month/year) for\n";
108   std::cout << "day =? ";
109   unsigned int day; std::cin >> day;
110   std::cout << "month =? ";
111   unsigned int month; std::cin >> month;
112   std::cout << "year =? ";
113   unsigned int year; std::cin >> year;
114
115   // check date
116   if (!is_date (day, month, year)) {
117     std::cout << "Illegal date.\n";
118     return 1;
119   }
120
```

```
121     // output weekday
122     print_weekday (weekday (day, month, year));
123
124     return 0;
125   }
```

Solution to Exercise 90. 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.C` 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.C
```

The directory after the `-I` is the one where the compiler will look for your include files. The path `IFM/math.h` in `math.C` 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.C -lmath -o callpow4
```

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

Solution to Exercise 91.

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).

```
#include <iostream>

// POST: return value is the integer nearest to x; if there are
//       two nearest integers, the one closer to 0 is chosen
int round (double x)
{
  int trunc = int(x); // rounds towards 0 by standard
  double error = x - trunc; // note: result is exact!
  if (error > 0.5)
    // x was positive, and trunc + 1 is nearer
    return trunc + 1;
  if (error < -0.5)
    // x was negative, and trunc - 1 is nearer
    return trunc - 1;
  // |error| <= 0.5, trunc is closest integer
  return trunc;
}

int main ()
{
  for (double d = -2; d <= 2; d += 0.25)
    std::cout << "closest integer to " << d << " is "
              << round (d) << "\n";

  return 0;
}
```

As fas as the integration into the library is concerned, please follow the procedure in the solution to Exercise 90 above.

Solution to Exercise 92. 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.

```
// Prog: power_cross_sums.C
// find the values a and b smaller than 100 such that
// a^b has maximum cross sum

#include <iostream>
#include "integer.h"

// POST: returns sum of decimal digits of n
ifm::integer cross_sum (ifm::integer n)
{
  // the straightforward method
  ifm::integer cross = 0;
  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    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 93.

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 94. 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.C
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 (unsigned int r, unsigned int c, unsigned int n)
29 {
30   // assert precondition
31   assert (grid[r][c][0] && n > 0 && grid[r][c][n]);
32   // go through row of (r,c)
33   for (unsigned int j=0; j<9; ++j)
34     grid[r][j][n] = false;
35   // go through column of (r,c)
36   for (unsigned int i=0; i<9; ++i)
37     grid[i][c][n] = false;
38   // go through box of (r,c)
39   unsigned int lr = r-r%3; // (lr, lc) is the lower left cell of
40   unsigned int lc = c-c%3; // this box
41   for (unsigned int i=lr; i<lr+3; ++i)
42     for (unsigned int j=lc; j<lc+3; ++j)
43       grid[i][j][n] = false;
44   // go through numbers
45   for (unsigned int m=0; m<10; ++m)
46     grid[r][c][m] = false;
```

```
47    // fill cell
48    grid[r][c][n] = true;
49  }
50
51  // POST: returns n > 0 if n is the unique candidate for cell (r,c) and
52  //       0 otherwise
53  int unique_number (unsigned int r, unsigned int c)
54  {
55    int n = 0;
56    bool found = false;
57    for (unsigned int m = 1; m<10; ++m)
58      if (grid[r][c][m]) {
59        if (found) return 0; // we've already seen a candidate
60        n = m;               // this is the first candidate
61        found = true;
62      }
63    return n;
64  }
65
66  // PRE:  cell (r,c) is empty, and n>0 is a candidate for (r,c)
67  // POST: returns true if and only if n is not a candidate for any
68  //       cell distinct from (r,c) within the same row
69  bool deducible_from_row (unsigned int r, unsigned int c, unsigned int n)
70  {
71    // assert preconditions
72    assert (grid[r][c][0] && n > 0 && grid[r][c][n]);
73    // go through the columns
74    for (int j=0; j<9; ++j)
75      if (j != c && grid[r][j][n]) return false; // candidate somewhere else
76    return true;
77  }
78
79  // PRE:  cell (r,c) is empty, and n>0 is a candidate for (r,c)
80  // POST: returns true if and only if n is not a candidate for any
81  //       cell distinct from (r,c) within the same column
82  bool deducible_from_column (unsigned int r, unsigned int c, unsigned int n)
83  {
84    // assert preconditions
85    assert (grid[r][c][0] && n > 0 && grid[r][c][n]);
86    // go through the rows
87    for (int i=0; i<9; ++i)
88      if (i != r && grid[i][c][n]) return false; // candidate somewhere else
89    return true;
90  }
91
92  // PRE:  cell (r,c) is empty, and n>0 is a candidate for (r,c)
93  // POST: returns true if and only if n is not a candidate for any
94  //       cell distinct from (r,c) within the same box
95  bool deducible_from_box (unsigned int r, unsigned int c, unsigned int n)
96  {
97    // assert preconditions
98    assert (grid[r][c][0] && n > 0 && grid[r][c][n]);
99    unsigned int lr = r-r%3; // (lr, lc) is the lower left cell of
100   unsigned int lc = c-c%3; // this box
101   // go through the box
102   for (int i=lr; i<lr+3; ++i)
103     for (int j=lc; j<lc+3; ++j)
104       if ( (i != r || j != c) && grid[i][j][n])
105         return false; // candidate somewhere else
106   return true;
107 }
108
109 // POST: returns true iff (r,c) is empty, n>0 is a candidate for (r,c), and
110 //         - n is the unique candidate for the cell (r,c), or
111 //         - n is deducible as the unique candidate from the row,
```

```
112  //           column, or box of (r,c)
113  bool deducible (unsigned int r, unsigned int c, unsigned int n)
114  {
115    if (grid[r][c][0] && n > 0 && grid[r][c][n])
116      return
117        (n == unique_number (r,c))      ||  // unique candidate for (r,c) ?
118        deducible_from_row (r, c, n)    ||
119        deducible_from_column (r, c, n) ||
120        deducible_from_box (r, c, n);
121    return false;
122  }
123
124  // POST: returns true if and only if there is an empty cell (r,c) and a
125  //       number n>0 such that n can be deduced to be the number to be
126  //       put into (r,c); if the return value is true, the grid is updated
127  //       by filling an empty cell with an accordingly deduced number
128  bool fill_cell()
129  {
130    for (unsigned int r=0; r<9; ++r)
131      for (unsigned int c=0; c<9; ++c)
132        for (unsigned int n=1; n<10; ++n)
133          if (deducible (r, c, n)) {
134            update_grid (r, c, n);
135            return true;
136          }
137    return false;
138  }
139
140  int main()
141  {
142    // set up empty grid
143    for (int r=0; r<9; ++r)
144      for (int c=0; c<9; ++c) {
145        for (int n=0; n<10; ++n)
146          grid[r][c][n] = true;
147      }
148
149    // input: 9 x 9 numbers in {0,..,9} rowwise (0 means no number)
150    unsigned int filled_cells = 0; // total number of filled cells
151    for (int r=0; r<9; ++r)
152      for (int c=0; c<9; ++c) {
153        int n;
154        std::cin >> n;
155        if (n > 0) {
156          update_grid (r, c, n);
157          ++filled_cells;
158        }
159      }
160
161    // main loop
162    while (fill_cell()) ++filled_cells;
163
164    // sudoku is solved if all cells could be filled
165    if (filled_cells == 81)
166      std::cout << "Sudoku is solved:\n";
167    else
168      std::cout << "Could only fill " << filled_cells << " cells:\n";
169    // output solution in 3 x 3 blocks (0 means no number deduced);
170    for (unsigned int lr=0; lr<9; lr+=3) {
171      for (unsigned int r=lr; r<lr+3; ++r) {
172        for (unsigned int lc=0; lc<9; lc+=3) {
173          for (unsigned int c=lc; c<lc+3; ++c)
174            std::cout << unique_number (r, c) << " "; // blank after number
175          std::cout << " "; // extra blank after every three columns
176        }
```

```
177         std::cout << "\n"; // extra line break after every third row
178       }
179       std::cout << "\n"; // line break after row
180     }
181   }
```