

Solution to Exercise 65.

- a) The program outputs 5 4 1 6 0. The first array element is always the first one to be output by `std::cout << *p`, because `p` is initialized with `a`, and due to array-to-pointer conversion, `p` then points to the first element of `a`. The assignment `p = a + *p` changes `p` into a pointer to the element of index `*p` in the array (see the paragraph on adding an integer to a pointer). Since `*p` is initially 5, we get the element of index 5 next, which is 4. The next element is the one of index 4 (1), followed by the ones of index 1 (6) and index 6 (0). At this point, `p` points again to the first element of the array, so the condition `p!=a` fails, and the loop terminates.
- b) The general structure is this: Let i_t be the index of the array element pointed to by `p` after t iterations. We have $i_0 = 0$, and $i_t = a[i_{t-1}]$ for $t > 0$. The program terminates as soon as $i_t = 0$ again for some $t > 0$. We must prove that this always happens.

Assume for contradiction that 0 does not appear for a second time, so that we have an infinite loop. Since there are only n possible index values, some index distinct from 0 must appear twice in the infinite sequence i_0, i_1, \dots . Let k be some value such that $i_k \neq 0$ already appears among i_0, \dots, i_{k-1} , and let $0 < \ell < k$ be such that $i_\ell = i_k$. By definition of the sequence, it follows that

$$a[i_{k-1}] = i_k = i_\ell = a[i_{\ell-1}],$$

where $k-1 \neq \ell-1$. But this gives the desired contradiction, since the array `a` was initialized with a sequence of pairwise distinct numbers.

Solution to Exercise 66. In a), we use

```
int* p = a+i; // address of element of index i
```

Note that

```
int* p = &a[i]; // address of element of index i
```

also works, *unless* $i = n$, since n is an out-of-bound index. A past-the-end pointer can therefore only be obtained from the first variant.

In b), we use pointer subtraction:

```
int i = p-a; // distance between *p and a[0] in the array
```

Solution to Exercise 67. We recycle the program `eratosthenes2.C` (Program 15); instead of maintaining the information whether a number has been crossed out, we maintain the information about the number of different prime divisors.

```
1 // Program: k_composite.C
2 // Calculate k-composite numbers in {2,...,n-1} using
3 // a variant of Eratosthenes' sieve.
```

```

4
5 #include <iostream>
6
7 int main()
8 {
9     // input of k
10    std::cout << "Compute k-composite numbers for k =? ";
11    unsigned int k;
12    std::cin >> k;
13
14    // input of n
15    std::cout << "Compute " << k
16            << "-composite numbers in {2,...,n-1} for n =? ";
17    unsigned int n;
18    std::cin >> n;
19
20    // definition and initialization: provides us with
21    // unsigned integers composition[0],..., composition[n-1]
22    unsigned int* composition = new unsigned int [n];
23    for (unsigned int i = 0; i < n; ++i)
24        composition[i] = 0; // no information yet
25
26    // computation and output
27    std::cout << k << "-composite numbers in {2,...,"
28            << n-1 << "}: \n";
29    for (unsigned int i = 2; i < n; ++i) {
30        if (composition[i] == 0) {
31            // i is prime: add 1 to composition number of all
32            // multiples (including i)
33            for (unsigned int m = i; m < n; m += i)
34                ++composition[m];
35        }
36        // now the composition number of i is up-to-date
37        if (composition[i] == k)
38            std::cout << i << " ";
39    }
40    std::cout << "\n";
41
42    delete[] composition; // free dynamic memory
43
44    return 0;
45 }

```

The following are the 8 different 7-composite numbers smaller than 1,000,000: 510510, 570570, 690690, 746130, 870870, 881790, 903210, 930930.

Solution to Exercise 68. One trick here is that computations with indices modulo 3 save us the signs $(-1)^{i+j}$.

```

1 // Prog: inverse_matrix.C
2 // read in a 3x3 matrix A, compute its inverse A^{-1},
3 // and output it along with A x A^{-1} as a crosscheck
4
5 #include <iostream>
6
7 int main()
8 {
9     // read in A (as a sequence of 9 numbers)
10    double a[3][3];
11    for (int i=0; i<3; ++i)
12        for (int j=0; j<3; ++j)
13            std::cin >> a[i][j];

```

```

14
15 // compute determinant of A via Sarrus' rule
16 double det = 0;
17 for (int i=0; i<3; ++i)
18     det += a[0][i] * a[1][(i+1)%3] * a[2][(i+2)%3]
19         - a[2][i] * a[1][(i+1)%3] * a[0][(i+2)%3];
20
21 // compute entries of A^{-1} through Cramer's rule
22 double a_inv[3][3];
23 for (int i=0; i<3; ++i)
24     for (int j=0; j<3; ++j)
25         a_inv[i][j] = (a[(j+1)%3][(i+1)%3] * a[(j+2)%3][(i+2)%3]
26             - a[(j+2)%3][(i+1)%3] * a[(j+1)%3][(i+2)%3]) / det;
27
28 // output A^{-1}
29 for (int i=0; i<3; ++i) {
30     for (int j=0; j<3; ++j)
31         std::cout << a_inv[i][j] << " ";
32     std::cout << "\n";
33 }
34
35 // crosscheck
36 for (int i=0; i<3; ++i) {
37     for (int j=0; j<3; ++j)
38         // output (A x A^{-1})_{ij}
39         std::cout <<
40             a[i][0]*a_inv[0][j]+
41             a[i][1]*a_inv[1][j]+
42             a[i][2]*a_inv[2][j] << " ";
43     std::cout << "\n";
44 }
45
46
47
48
49
50 return 0;
51
52 }

```

Solution to Exercise 69.

```

1 // Program: read_array.C
2 // read a sequence of n numbers into an array
3 #include <iostream>
4
5 int main()
6 {
7     // input of n
8     unsigned int n;
9     std::cin >> n;
10
11     // dynamically allocate array
12     int* a = new int[n];
13
14     // read into the array
15     for (int i=0; i<n; ++i) std::cin >> a[i];
16
17     // output what we have
18     for (int i=0; i<n; ++i) std::cout << a[i] << " ";
19     std::cout << "\n";
20
21     // delete array

```

```

22 delete[] a;
23
24 return 0;
25
26 }

```

Solution to Exercise 70.

```

1 // Program: sort_array.C
2 // read a sequence of n numbers into an array,
3 // sort them, and output the sorted sequence
4 #include <iostream>
5
6 int main()
7 {
8     // input of n
9     unsigned int n;
10    std::cin >> n;
11
12    // dynamically allocate array
13    int* a = new int[n];
14
15    // read into the array
16    for (int i=0; i<n; ++i) std::cin >> a[i];
17
18    // sort array: in round i=0,...,n-2 we find
19    // the smallest element in a[i],...,a[n-1] and
20    // interchange it with a[i]
21    for (int i=0; i<n-1; ++i) {
22        // find minimum in a[i],...,a[n-1]
23        int i_min = n-1; // index of minimum
24        for (int j=i; j<n-1; ++j)
25            if (a[j] < a[i_min]) i_min = j;
26        // interchange a[i] with a[i_min]
27        int h = a[i]; a[i] = a[i_min]; a[i_min] = h;
28    }
29
30    // output sorted sequence
31    for (int i=0; i<n; ++i) std::cout << a[i] << " ";
32    std::cout << "\n";
33
34    // delete array
35    delete[] a;
36
37    return 0;
38
39 }

```

Solution to Exercise 71.

```

1 // Program: cycles.C
2 // read a sequence of n numbers into an array; if the sequence
3 // encodes a permutation of {0,...,n-1}, output its cycle
4 // decomposition
5
6 #include <iostream>
7
8 int main()
9 {
10    // input of n

```

```

11  unsigned int n;
12  std::cin >> n;
13
14  // dynamically allocate array for the numbers, and a
15  // second array of booleans to keep track of which
16  // numbers are present
17  int* a = new int[n];
18  bool* present = new bool[n];
19
20  // initialize array present
21  for (bool* r=present; r<present+n; ++r) *r = false;
22
23  // read number into the array and remember that it was read
24  for (int* p=a; p<a+n; ++p) {
25      std::cin >> *p;
26      if (*p >= 0 && *p < n)
27          present[*p] = true;
28  }
29
30  // check whether we have read all numbers in {0,...,n-1}
31  bool ok = true;
32  for (bool* r=present; r<present+n; ++r)
33      if (!*r) {
34          std::cout << "input sequence does not encode a permutation.\n";
35          ok = false;
36          break;
37      }
38
39  if (ok) {
40      // do the cycle decomposition. Here we reuse the array present
41      // and remove from it all numbers that we have already put into
42      // some cycle
43      std::cout << "cycle decomposition is ";
44      int next = 0; // next number not yet put into a cycle
45      while (next < n) {
46          // output cycle starting with next; we must come back to next
47          // below: assuming we would come back to some other element
48          // on the cycle, that element would have two preimages under
49          // pi, a contradiction
50          int first = next;
51          std::cout << "( ";
52          do {
53              std::cout << next << " ";
54              present[next] = false;
55              next = a[next]; // next -> pi(next)
56          } while (next != first);
57          std::cout << ") ";
58
59          // find start element of next cycle
60          while (!present[next]) ++next;
61      }
62      std::cout << "\n";
63  }
64  // delete arrays
65  delete[] present;
66  delete[] a;
67
68  return 0;
69
70 }

```

Solution to Exercise 72. Let $s = a \dots ab$ (i.e. $m - 1$ a's followed by one b); let $t = a \dots a$ (i.e. n a's). Then the algorithm must always go through all m characters of any window

in order to find the mismatch with b at the last position. Since it in total processes $n - m + 1$ windows $\{1, \dots, m\}$ up to $\{n - m + 1, \dots, n\}$, the number of comparisons is $m(n - m + 1)$.

Solution to Exercise 73.

```

1 // Program: threedim_array.C
2 // iterate over a multidimensional array
3 #include <iostream>
4
5 int main()
6 {
7     int a[4][2][3] =
8     { // the 4 elements of a:
9     { // the 2 elements of a[0]:
10         {2, 4, 5}, // the three elements of a[0][0]
11         {4, 6, 7} // the three elements of a[0][1]
12     },
13     { // the 2 elements of a[1]:
14         {1, 5, 9}, // the three elements of a[1][0]
15         {4, 6, 1} // the three elements of a[1][1]
16     },
17     { // the 2 elements of a[2]:
18         {5, 9, 0}, // the three elements of a[2][0]
19         {1, 5, 3} // the three elements of a[2][1]
20     },
21     { // the 2 elements of a[3]:
22         {6, 7, 7}, // the three elements of a[3][0]
23         {7, 8, 5} // the three elements of a[3][1]
24     }
25 };
26
27 for (int (*i)[2][3] = a; i < a + 4; ++i) {
28     // i (pointer to int[2][3]) points to a[0], ..., a[3]
29     // *i therefore assumes the values a[0], ..., a[3]
30     for (int (*j)[3] = *i; j < *i + 2; ++j) {
31         // j (pointer to int[3]) points to a[i][0], ..., a[i][1]
32         // *j therefore assumes the values a[i][0], ..., a[i][1]
33         for (int* k = *j; k < *j + 3; ++k)
34             // k (pointer to int) points to a[i][j][0], ..., a[i][j][2]
35             // *k therefore assumes the values a[i][j][0], ..., a[i][j][2]
36             std::cout << *k << " ";
37         std::cout << "\n";
38     }
39     std::cout << "\n";
40 }
41
42 return 0;
43
44 }
```

Some people might be tempted by the following kind of approach:

```

1 // Program: threedim_array.C
2 // (erroneously) iterate over a multidimensional array
3 #include <iostream>
4
5 int main()
6 {
7     int a[4][2][3] =
8     { // the 4 elements of a:
```

```

9      { // the 2 elements of a[0]:
10        {2, 4, 5}, // the three elements of a[0][0]
11        {4, 6, 7} // the three elements of a[0][1]
12      },
13      { // the 2 elements of a[1]:
14        {1, 5, 9}, // the three elements of a[1][0]
15        {4, 6, 1} // the three elements of a[1][1]
16      },
17      { // the 2 elements of a[2]:
18        {5, 9, 0}, // the three elements of a[2][0]
19        {1, 5, 3} // the three elements of a[2][1]
20      },
21      { // the 2 elements of a[3]:
22        {6, 7, 7}, // the three elements of a[3][0]
23        {7, 8, 5} // the three elements of a[3][1]
24      }
25    };
26
27    int* p = a[0][0]; // pointer to a[0][0][0]
28    for (int i=0; i<24; ++i)
29      std::cout << *p++ << " ";
30
31    std::cout << "\n";
32
33    return 0;
34 }

```

This indeed does not contradict anything written in Section 2.6; in particular, any operation `++p` (which reduces to `p+1` plus an assignment) has the property that both `p` as well as `p+1` point to elements (or past the end) of the *same* array. However, that array changes during the increment. For the first three `++p`'s, it's the array `a[0][0]`, for the second three, it's `a[0][1]`, and so on. But changing the array during pointer increment is not allowed by the C++ standard; in fact, the standard allows (by not forbidding it) implementations of pointer arithmetic that perform bounds checking. Such an implementation might give you a runtime error if you try to increment `p` further than past the end of `a[0][0]`, the array on which the pointer logically “lives”.

Solution to Exercise 74. The trick is to use characters (which have integral values) directly as array indices.

```

1 // Program: frequencies.C
2 // output frequencies of the letters in an input text
3
4 #include <iostream>
5
6 int main ()
7 {
8     // array for number of occurrences of every ASCII character
9     int frequency[128];
10    for (int i=0; i<128; ++i) frequency[i] = 0;
11
12    // now scan the text
13    char c; // next character
14    unsigned int total = 0; // text length
15    while (std::cin >> c) {
16        ++total;
17        ++frequency[c];
18    }

```

```

19
20 // output
21 unsigned int letters = 0; // number of letters
22 std::cout << "Frequencies: \n";
23 for (char c = 'a'; c <= 'z'; ++c) {
24     int f = frequency[c] + frequency[c-32]; // lower + upper case c
25     letters += f;
26     std::cout << c << ":      " << f << " of " << total << "\n";
27 }
28 std::cout << "Other: " << total-letters << " of " << total << "\n";
29
30 return 0;
31 }

```

Solution to Exercise 75. Here is a solution. We initially dynamically allocate an array of length $n = 1$ (pointed to by a pointer a), and whenever the next sequence element wouldn't fit anymore, we replace the array by a new one of length $2n$. For this, we first dynamically allocate a helper array, copy the contents of the current array into the helper array, delete the current array and then let a point to the newly allocated helper array.

```

1 // Program: read_array.C
2 // read a sequence of numbers into an array
3 #include <iostream>
4
5 int main()
6 {
7     int n = 1; // current array size
8     int k = 0; // number of elements read so far
9
10    // dynamically allocate array
11    int* a = new int[n];
12
13    // read into the array
14    while (std::cin >> a[k]) {
15        if (++k == n) {
16            // next element wouldn't fit; replace the array a by
17            // a new one of twice the size
18            int* b = new int[n*=2]; // get pointer to new array
19            for (int i=0; i<k; ++i) // copy old array to new one
20                b[i] = a[i];
21            delete[] a; // delete old array
22            a = b; // let a point to new array
23        }
24    }
25
26    // output the first k elements
27    for (int i=0; i<k; ++i) std::cout << a[i] << " ";
28    std::cout << "\n";
29
30    // delete array
31    delete[] a;
32
33    return 0;
34
35 }

```

This is space and time efficient. The constant of proportionality in (i) is 3: whenever we grow the array (and these are the points in time where the ratio between memory

cells in use and k is largest), we allocate a new array of length $2n = 2k$, in addition to the one of length k that we already have. This means that we have $3k$ memory cells in use at that time.

The constant of proportionality in (ii) is 3 as well. To see this, let us consider the situation after an execution of the while loop, where k is the number of elements read so far. There have been k assignments to array elements in the loop's condition, and some additional assignments during the copying of old to new array. Such assignments took place when the number of elements currently read was a power of two less or equal to k , and the number of these additional assignments was exactly the power of two in question. Since the sum of all powers of two less or equal to k is at most $k + k/2 + k/4 + \dots \leq 2k$, the total number of assignments is bounded by $k + 2k = 3k$.

Solution to Exercise 76. Here is the faster program.

```

1  #include <iostream>
2  #include <cassert>
3
4  int main()
5  {
6      // read floor dimensions
7      int n; std::cin >> n; // number of rows
8      int m; std::cin >> m; // number of columns
9
10     // dynamically allocate twodimensional array of dimensions
11     // (n+2) x (m+2) to hold the floor plus extra walls around
12     int** floor = new int*[n+2];
13     for (int r=0; r<n+2; ++r)
14         floor[r] = new int[m+2];
15
16     // we need another two arrays for storing row and column
17     // indices of already labeled cells;
18     int* labeled_r = new int[n*m];
19     int* labeled_c = new int[n*m];
20
21     // in order to search for new cells to be labeled, we
22     // always start from the first labeled cell whose neighbors
23     // have not been looked at yet;
24     int next_l = 0; // index of this cell
25
26     // whenever we label a cell, we append it to the list of
27     // labeled cells; that way, the cells are ordered by label
28     // in the list
29     int last_l = 0; // one plus index of last cell in this list
30
31     // target coordinates, set upon reading 'T'
32     int tr = 0;
33     int tc = 0;
34
35     // assign initial floor values from input:
36     // source:      'S'  ->    0 (source reached in 0 steps)
37     // target:      'T'  ->   -1 (number of steps still unknown)
38     // wall:        'X'  ->   -2
39     // empty cell: '-'  ->   -1 (number of steps still unknown)
40     for (int r=1; r<n+1; ++r)
41         for (int c=1; c<m+1; ++c) {
42             char entry = '-';
43             std::cin >> entry;
44             if (entry == 'S') {
```

```

45     floor[r][c] = 0;
46     labeled_r[last_l] = r;
47     labeled_c[last_l] = c;
48     ++last_l;
49 }
50 else if (entry == 'T') floor[tr = r][tc = c] = -1;
51 else if (entry == 'X') floor[r][c] = -2;
52 else if (entry == '-') floor[r][c] = -1;
53 }
54
55 // add surrounding walls
56 for (int r=0; r<n+2; ++r)
57     floor[r][0] = floor[r][m+1] = -2;
58 for (int c=0; c<m+2; ++c)
59     floor[0][c] = floor[n+1][c] = -2;
60
61 // main loop: process next labeled cell until done
62 while (next_l != last_l) {
63     int r = labeled_r[next_l];
64     int c = labeled_c[next_l];
65     int i = floor[r][c];
66     assert (i >= 0);
67     // label the unlabeled neighbors by i+1
68     for (int rr = r-1; rr <= r+1; ++rr)
69         for (int cc = c-1; cc <= c+1; ++cc)
70             if ( (rr == r || cc == c) && floor[rr][cc] == -1) {
71                 // we have a neighbor, and it's not labeled yet
72                 floor[rr][cc] = i+1;
73                 labeled_r[last_l] = rr;
74                 labeled_c[last_l] = cc;
75                 ++last_l;
76             }
77     ++next_l;
78 }
79
80 // mark shortest path from source to target (if there is one)
81 int r = tr; int c = tc; // start from target
82 while (floor[r][c] > 0) {
83     int d = floor[r][c] - 1; // distance one less
84     floor[r][c] = -3; // mark cell as being on shortest path
85     // go to some neighbor with distance d
86     if (floor[r-1][c] == d) --r;
87     else if (floor[r+1][c] == d) ++r;
88     else if (floor[r][c-1] == d) --c;
89     else ++c; // (floor[r][c+1] == d)
90 }
91
92 // print floor with shortest path
93 for (int r=1; r<n+1; ++r) {
94     for (int c=1; c<m+1; ++c)
95         if (floor[r][c] == 0)    std::cout << 'S';
96         else if (r == tr && c == tc) std::cout << 'T';
97         else if (floor[r][c] == -3) std::cout << 'o';
98         else if (floor[r][c] == -2) std::cout << 'X';
99         else                    std::cout << '-';
100     std::cout << "\n";
101 }
102
103 // delete dynamically allocated arrays
104 delete[] labeled_c;
105 delete[] labeled_r;
106 for (int r=0; r<n+2; ++r)
107     delete[] floor[r];
108 delete[] floor;
109

```

```

110     return 0;
111 }

```

Solution to Exercise 77. The idea is to try all possible combinations of a, b, c within prespecified ranges. In order to make this fast, some tricks are needed, though. First of all, we use Eratosthenes' Sieve in order to precompute the information whether a given number that may arise as $|an^2 + bn + c|$ is prime. Then we use another array to mark the primes that we have seen in a run of primes; the trick here is to delete the markers again in an efficient way. The following program discovers a quadratic polynomial of Euler quality 45, namely

$$36n^2 - 810n + 2753.$$

```

1 // Program: euler_prime.C
2 // Finds a,b,c within specified bounds such that
3 // the formula |an^2 + bn + c| produces the largest
4 // number of distinct consecutive primes, starting
5 // with n = 0
6
7 #include <iostream>
8 #include <cassert>
9
10 int main ()
11 {
12     // by multiplying with -1, if necessary, we may assume c >= 0
13     const int arange = 100;      // a in {-arange+1,...,arange-1}
14     const int brange = 1000;    // b in {-brange+1,...,brange-1}
15     const int crange = 10000;   // c in {2,...,crange-1}
16
17     // first, compute all primes in the set {an^2 + bn + c} where a, b, c
18     // run through their ranges and n is smaller than 100 (we're not
19     // searching for longer runs of primes here)
20     const int max_elem = arange * 10000 + brange * 100 + crange;
21
22     // run Eratosthenes' sieve
23     bool prime[max_elem];
24     for (int i = 2; i < max_elem; ++i)
25         prime[i] = true;
26     for (int i = 2; i < max_elem; ++i)
27         if (prime[i]) {
28             // cross out all proper multiples of i
29             for (int m = 2*i; m < max_elem; m += i)
30                 prime[m] = false;
31         }
32
33     // now search for the best a, b, c
34     int best_run = 0;
35     int best_a = arange;
36     int best_b = brange;
37     int best_c = crange;
38
39     // array to keep track of primes we have already seen in a run
40     bool seen_before[max_elem];
41     for (int i = 2; i < max_elem; ++i)
42         seen_before[i] = false;
43
44     // loop over all candidate triples (a,b,c)
45     for (int c = 2; c < crange; ++c) {

```

```

46     if (!prime[c]) continue; // not even a prime for n = 0
47     for (int a = -arange+1; a < arange; ++a)
48         for (int b = -brange+1; b < brange; ++b) {
49             // evaluate elem = an^2 + bn + c for n=0,1,...
50             int n = 0;
51             int elem = c;
52             for (; n < 100; ++n) {
53                 int abs_elem = elem < 0 ? -elem: elem;
54                 if (abs_elem < 2 || !prime[abs_elem]) break; // not a prime
55                 if (seen_before[abs_elem]) break;           // repeated prime
56                 seen_before[abs_elem] = true;               // new prime
57                 // update element
58                 elem += a * (2*n + 1) + b;
59             }
60             // now we have seen a run of n primes (for 0,...,n-1)
61             if (n > best_run) {
62                 best_run = n;
63                 best_a = a;
64                 best_b = b;
65                 best_c = c;
66             }
67             // remove the "seen_before" markers for the next run
68             elem = c;
69             for (int k=0; k<n; ++k) {
70                 int abs_elem = elem < 0 ? -elem: elem;
71                 seen_before[abs_elem] = false;
72                 elem += a * (2*k + 1) + b;
73             }
74         }
75     }
76
77     std::cout << "Best a      = " << best_a << ".\n";
78     std::cout << "Best b      = " << best_b << ".\n";
79     std::cout << "Best c      = " << best_c << ".\n";
80     std::cout << "Euler quality = " << best_run << ".\n";
81
82     return 0;
83 }

```

Solution to Exercise 78.

```

1 // Prog: xbm.C
2 // includes an xbm file and outputs the xbm-file that corresponds
3 // to the image rotated by 90 degrees
4
5 #include<iostream>
6 #include<cassert>
7 #include "test.xbm"
8 #define width test_width
9 #define height test_height
10 #define bits test_bits
11
12 int main()
13 {
14     // array for mapping from {0,...,15} to {'0',...,'f'}
15     char hex[] = {'0', '1', '2', '3', '4', '5', '6', '7',
16                 '8', '9', 'a', 'b', 'c', 'd', 'e', 'f'};
17
18     // output header for rotated image
19     std::cout << "#define rotated_width " << height << "\n";
20     std::cout << "#define rotated_height " << width << "\n";
21     std::cout << "static char rotated_bits[] = {\n";
22

```

```

23 // go through the pixels columnwise (from right to left),
24 // and within each column, proceed from top to bottom
25 unsigned int columns = (width+7)/8; // bytes in row (original)
26 unsigned int rows = (height+7)/8; // bytes in column (rotated)
27 bool comma = false; // not before first byte
28 for (int c=columns-1; c>=0; --c) {
29 // go through columns from right to left
30 for (int d=128; d>0; d/=2) {
31 // for d = 2^j, we are in column c+j
32 for (int r=0; r<rows; ++r) {
33 // go through rows from top to bottom and build
34 // up one byte from any 8 pixels (least significant
35 // digit comes from pixel in first of these rows)
36 int byte = 0;
37 for (int i=7; i>=0; --i) {
38 // we are in row 8r+i
39 int pixel;
40 if (8*r+i >= height)
41 // non-existing row, fill up with zeros
42 pixel = 0;
43 else
44 // get pixel in row 8r+i and column c+j
45 pixel = (unsigned char)(bits[(8*r+i)*columns+c])/d%2;
46 byte = 2*byte+pixel;
47 }
48 if (comma) std::cout << ", "; comma = true;
49 std::cout << "0x" << hex[byte/16] << hex[byte%16];
50 }
51 }
52 std::cout << "\n";
53 }
54 std::cout << "]\n";
55
56 return 0;
57
58 }

```
