B.6 Arrays and pointers

Solution to Exercise 76.

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 $i_0, i_1, \ldots$. Let k be some value such that $i_k \neq 0$ already appears among $i_0, \ldots, i_{k-1}$, 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 77. 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 78.

Problem 1:
The pointer `b` has been declared as a constant, but later it is incremented (`*b++`). This won't compile. Fix: remove the `const`. Now the program at least compiles...

Problem 2:
In the first loop, the range that p points to is too large by one address. Fix: `p <= a+7` should be `p < a+7`.

Problem 3
A similar problem also appears in the second loop. Inside the loop a and b are accessed by one too many values for index i. Fix: `i<=7` should be `i<7`.

Problem 4
The cross-check in the second loop goes wrong, because b doesn't point to the beginning of the dynamically created array any more. Fix: copy the pointer b to a pointer d before the first loop, and iterate over the pointer d in the second loop. Then, we can also reintroduce the const for b.

Problem 5
The wrong type of delete-operator is used. Fix: delete should be delete[].

If all these fixed are applied, there is still one logical const missing for const-correctness. The loop pointer p should have underlying type const int to reflect the fact that the loop does not change the array a that it loops over.

Here is the resulting correct program.

```
#include<iostream>

int main()
{
  int a[7] = {0, 6, 5, 3, 2, 4, 1}; // static array
  int* const b = new int[7];
  int* d = b;

  // copy a into b using pointers
  for (const int* p = a; p < a+7; ++p)
    *d++ = *p;

  // cross-check with random access
  for (int i = 0; i < 7; ++i)
    if (a[i] != b[i])
      std::cout << "Oops, copy error...\n";

  delete[] b;

  return 0;
}
```

Solution to Exercise 79. We recycle the program eratosthenes2.cpp (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.

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

#include <iostream>

int main()
{
```

```
  // input of k
  std::cout << "Compute k-composite numbers for k =? ";
  unsigned int k;
  std::cin >> k;

  // input of n
  std::cout << "Compute " << k
            << "-composite numbers in {2,...,n-1} for n =? ";
  unsigned int n;
  std::cin >> n;

  // definition and initialization: provides us with
  // unsigned integers composition[0],..., composition[n-1]
  unsigned int* const composition = new unsigned int [n];
  for (unsigned int i = 0; i < n; ++i)
    composition[i] = 0; // no information yet

  // computation and output
  std::cout << k << "-composite numbers in {2,...,"
            << n-1 << "}:\n";
  for (unsigned int i = 2; i < n; ++i) {
    if (composition[i] == 0) {
      // i is prime: add 1 to composition number of all
      // multiples (including i)
      for (unsigned int m = i; m < n; m += i)
        ++composition[m];
    }
    // now the composition number of i is up-to-date
    if (composition[i] == k)
      std:: cout << i << " ";
  }
  std::cout << "\n";

  delete[] composition;                  // free dynamic memory

  return 0;
}
```

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

```
// Prog: inverse_matrix.cpp
// read in a 3x3 matrix A, compute ins inverse A^{-1},
// and output it along with A x A^{-1} as a crosscheck

#include<iostream>

int main()
{
  // read in A (as a sequence of 9 numbers)
  double a[3][3];
  for (int i=0; i<3; ++i)
    for (int j=0; j<3; ++j)
      std::cin >> a[i][j];

  // compute determinant of A via Sarrus' rule
  double det = 0;
  for (int i=0; i<3; ++i)
    det += a[0][i] * a[1][(i+1)%3] * a[2][(i+2)%3]
```

```
      - a[2][i] * a[1][(i+1)%3] * a[0][(i+2)%3];

  // compute (and output) entries of A^{-1} through Cramer's rule
  std::cout << "A^-1 = \n";
  double a_inv[3][3];
  for (int i=0; i<3; ++i) {
    for (int j=0; j<3; ++j)
      std::cout
        << (
            a_inv[i][j] = ( (a[(j+1)%3][(i+1)%3]*a[(j+2)%3][(i+2)%3]) -
                            (a[(j+1)%3][(i+2)%3]*a[(j+2)%3][(i+1)%3]) ) / det)
        << " ";
    std::cout << "\n";
  }

  // crosscheck
  std::cout << "A * A^-1 = \n";
  for (int i=0; i<3; ++i) {
    for (int j=0; j<3; ++j)
      // output (A x A^{-1})_{ij}
      std::cout <<
        a[i][0]*a_inv[0][j]+
        a[i][1]*a_inv[1][j]+
        a[i][2]*a_inv[2][j] << " ";
    std::cout << "\n";
  }

  return 0;

}
```

Solution to Exercise 81.

```
// Program: read_array.cpp
// read a sequence of n numbers into an array
#include <iostream>

int main()
{
  // input of n
  unsigned int n;
  std::cin >> n;

  // dynamically allocate array
  int* const a = new int[n];

  // read into the array
  for (int i=0; i<n; ++i) std::cin >> a[i];

  // output what we have
  for (int i=0; i<n; ++i) std::cout << a[i] << " ";
  std::cout << "\n";

  // delete array
  delete[] a;

  return 0;

}
```

Solution to Exercise 82.

```
// Program: sort_array.cpp
// read a sequence of n numbers into an array,
// sort them, and output the sorted sequence
#include <iostream>

int main()
{
  // input of n
  unsigned int n;
  std::cin >> n;

  // dynamically allocate array
  int* const a = new int[n];

  // read into the array
  for (int i=0; i<n; ++i) std::cin >> a[i];

  // sort array: in round i=0,...,n-2 we find
  // the smallest element in a[i],...,a[n-1] and
  // interchange it with a[i]
  for (int i=0; i<n-1; ++i) {
    // find minimum in a[i],...,a[n-1]
    int i_min = n-1; // index of minimum
    for (int j=i; j<n-1; ++j)
      if (a[j] < a[i_min]) i_min = j;
    // interchange a[i] with a[i_min]
    const int h = a[i]; a[i] = a[i_min]; a[i_min] = h;
  }

  // output sorted sequence
  for (int i=0; i<n; ++i) std::cout << a[i] << " ";
  std::cout << "\n";

  // delete array
  delete[] a;

  return 0;

}
```

Solution to Exercise 83.

```
// Program: cycles.cpp
// read a sequence of n numbers into an array; if the sequence
// encodes a permutation of {0,...,n-1}, output its cycle
// decomposition

#include <iostream>

int main()
{
  // input of n
  unsigned int n;
  std::cin >> n;

  // dynamically allocate array for the numbers, and a
  // second array of booleans to keep track of which
  // numbers are present
  int* const a = new int[n];
  bool* const present = new bool[n];

  // initialize array present
```

```
  for (bool* r=present; r<present+n; ++r) *r = false;

  // read number into the array and remember that it was read
  for (int* p=a; p<a+n; ++p) {
    std::cin >> *p;
    if (*p >= 0 && *p < n)
      present[*p] = true;
  }

  // check whether we have read all numbers in {0,...,n-1}
  bool ok = true;
  for (bool* r=present; r<present+n; ++r)
    if (!*r) {
      std::cout << "input sequence does not encode a permutation.\n";
      ok = false;
      break;
    }

  if (ok) {
    // do the cycle decomposition. Here we reuse the array present
    // and remove from it all numbers that we have already put into
    // some cycle
    std::cout << "cycle decomposition is ";
    int next = 0; // next number not yet put into a cycle
    while (next < n) {
      // output cycle starting with next; we must come back to next
      // below: assuming we would come back to some other element
      // on the cycle, that element would have two preimages under
      // pi, a contradiction
      const int first = next;
      std::cout << "( ";
      do {
        std::cout << next << " ";
        present[next] = false;
        next = a[next]; // next -> pi(next)
      } while (next != first);
      std::cout << ") ";

      // find start element of next cycle
      while (!present[next]) ++next;
    }
    std::cout << "\n";
  }
  // delete arrays
  delete[] present;
  delete[] a;

  return 0;

}
```

Solution to Exercise 84. 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 85.

```
// Program: threedim_array.cpp
```

```
// iterate over a multidimensional array
#include <iostream>

int main()
{
  int a[4][2][3] =
    { // the 4 elements of a:
      { // the 2 elements of a[0]:
        {2, 4, 5}, // the three elements of a[0][0]
        {4, 6, 7}  // the three elements of a[0][1]
      },
      { // the 2 elements of a[1]:
        {1, 5, 9}, // the three elements of a[1][0]
        {4, 6, 1}  // the three elements of a[1][1]
      },
      { // the 2 elements of a[2]:
        {5, 9, 0}, // the three elements of a[2][0]
        {1, 5, 3}  // the three elements of a[2][1]
      },
      { // the 2 elements of a[3]:
        {6, 7, 7}, // the three elements of a[3][0]
        {7, 8, 5}  // the three elements of a[3][1]
      }
    };

  for (const int (*i)[2][3] = a; i < a + 4; ++i) {
    // i (pointer to int[2][3]) points to a[0],...,a[3]
    // *i therefore assumes the values a[0],...,a[3]
    for (const int (*j)[3] = *i; j < *i + 2; ++j) {
      // j (pointer to int[3]) points to a[i][0],...,a[i][1]
      // *j therefore assumes the values a[i][0],...,a[i][1]
      for (const int* k = *j; k < *j + 3; ++k)
        // k (pointer to int) points to a[i][j][0],...a[i][j][2]
        // *k therefore assumes the values a[i][j][0],...a[i][j][2]
        std::cout << *k << " ";
      std::cout << "\n";
    }
    std::cout << "\n";
  }

  return 0;

}
```

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

```
// Program: threedim_array.cpp
// (erroneously) iterate over a multidimensional array
#include <iostream>

int main()
{
  int a[4][2][3] =
    { // the 4 elements of a:
      { // the 2 elements of a[0]:
        {2, 4, 5}, // the three elements of a[0][0]
        {4, 6, 7}  // the three elements of a[0][1]
      },
      { // the 2 elements of a[1]:
        {1, 5, 9}, // the three elements of a[1][0]
        {4, 6, 1}  // the three elements of a[1][1]
      },
      { // the 2 elements of a[2]:
```

```
        {5, 9, 0}, // the three elements of a[2][0]
        {1, 5, 3}  // the three elements of a[2][1]
      },
      { // the 2 elements of a[3]:
        {6, 7, 7}, // the three elements of a[3][0]
        {7, 8, 5}  // the three elements of a[3][1]
      }
    };

  const int* p = a[0][0];         // pointer to a[0][0][0]
  for (int i=0; i<24; ++i)
    std::cout << *p++ << " ";

  std::cout << "\n";

  return 0;
}
```

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

```
// Program: frequencies.cpp
// output frequencies of the letters in an input text

#include<iostream>

int main ()
{
  // array for number of occurences of every ASCII character
  int frequency[128];
  for (int i=0; i<128; ++i) frequency[i] = 0;

  // now scan the text
  char c;                     // next character
  unsigned int total = 0;     // text length
  while (std::cin >> c) {
    ++total;
    ++frequency[c];
  }

  // output
  unsigned int letters = 0; // number of letters
  std::cout << "Frequencies: \n";
  for (char c = 'a'; c <= 'z'; ++c) {
    const int f = frequency[c] + frequency[c-32]; // lower + upper case c
    letters += f;
    std::cout << c << ":     " << f << " of " << total << "\n";
  }
```

```
  std::cout << "Other: " << total-letters << " of " << total << "\n";

  return 0;
}
```

Solution to Exercise 87. 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.

```
// Program: read_array.cpp
// read a sequence of numbers into an array
#include <iostream>

int main()
{
  int n = 1; // current array size
  int k = 0; // number of elements read so far

  // dynamically allocate array
  int* a = new int[n]; // this time, a is NOT a constant

  // read into the array
  while (std::cin >> a[k]) {
    if (++k == n) {
      // next element wouldn't fit; replace the array a by
      // a new one of twice the size
      int* b = new int[n*=2];  // get pointer to new array
      for (int i=0; i<k; ++i)  // copy old array to new one
        b[i] = a[i];
      delete[] a;              // delete old array
      a = b;                   // let a point to new array
    }
  }

  // output the first k elements
  for (int i=0; i<k; ++i) std::cout << a[i] << " ";
  std::cout << "\n";

  // delete array
  delete[] a;

  return 0;

}
```

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 $k + k/2 + k/4 + \cdots \leq 2k$, the total number of assignments is bounded by $k + 2k = 3k$.

Solution to Exercise 88. Here is the faster program.

```
#include<iostream>
#include<cassert>

int main()
{
  // read floor dimensions
  int n; std::cin >> n; // number of rows
  int m; std::cin >> m; // number of columns

  // dynamically allocate twodimensional array of dimensions
  // (n+2) x (m+2) to hold the floor plus extra walls around
  int** const floor = new int*[n+2];
  for (int r=0; r<n+2; ++r)
    floor[r] = new int[m+2];

  // we need another two arrays for storing row and column
  // indices of already labeled cells;
  int* const labeled_r = new int[n*m];
  int* const labeled_c = new int[n*m];

  // in order to search for new cells to be labeled, we
  // always start from the first labeled cell whose neighbors
  // have not been looked at yet;
  int next_l = 0; // index of this cell

  // whenever we label a cell, we append it to the list of
  // labeled cells; that way, the cells are ordered by label
  // in the list
  int last_l = 0; // one plus index of last cell in this list

  // target coordinates, set upon reading 'T'
  int tr = 0;
  int tc = 0;

  // assign initial floor values from input:
  // source:     'S'  ->     0 (source reached in 0 steps)
  // target:     'T'  ->    -1 (number of steps still unknown)
  // wall:       'X'  ->    -2
  // empty cell: '-'  ->    -1 (number of steps still unknown)
  for (int r=1; r<n+1; ++r)
    for (int c=1; c<m+1; ++c) {
      char entry = '-';
      std::cin >> entry;
      if      (entry == 'S') {
        floor[r][c] = 0;
        labeled_r[last_l] = r;
        labeled_c[last_l] = c;
        ++last_l;
      }
      else if (entry == 'T') floor[tr = r][tc = c] = -1;
      else if (entry == 'X') floor[r][c] = -2;
      else if (entry == '-') floor[r][c] = -1;
    }
```

```

  // add surrounding walls
  for (int r=0; r<n+2; ++r)
    floor[r][0] = floor[r][m+1] = -2;
  for (int c=0; c<m+2; ++c)
    floor[0][c] = floor[n+1][c] = -2;

  // main loop: process next labeled cell until done
  while (next_l != last_l) {
    const int r = labeled_r[next_l];
    const int c = labeled_c[next_l];
    const int i = floor[r][c];
    assert (i >= 0);
    // label the unlabeled neighbors by i+1
    for (int rr = r-1; rr <= r+1; ++rr)
      for (int cc = c-1; cc <= c+1; ++cc)
        if ( (rr == r || cc == c) && floor[rr][cc] == -1) {
          // we have a neighbor, and it's not labeled yet
          floor[rr][cc] = i+1;
          labeled_r[last_l] = rr;
          labeled_c[last_l] = cc;
          ++last_l;
        }
    ++next_l;
  }

  // mark shortest path from source to target (if there is one)
  int r = tr; int c = tc; // start from target
  while (floor[r][c] > 0) {
    const int d = floor[r][c] - 1; // distance one less
    floor[r][c] = -3; // mark cell as being on shortest path
    // go to some neighbor with distance d
    if      (floor[r-1][c] == d) --r;
    else if (floor[r+1][c] == d) ++r;
    else if (floor[r][c-1] == d) --c;
    else                         ++c; // (floor[r][c+1] == d)
  }

  // print floor with shortest path
  for (int r=1; r<n+1; ++r) {
    for (int c=1; c<m+1; ++c)
      if      (floor[r][c] == 0)   std::cout << 'S';
      else if (r == tr && c == tc) std::cout << 'T';
      else if (floor[r][c] == -3)  std::cout << 'o';
      else if (floor[r][c] == -2)  std::cout << 'X';
      else                         std::cout << '-';
    std::cout << "\n";
  }

  // delete dynamically allocated arrays
  delete[] labeled_c;
  delete[] labeled_r;
  for (int r=0; r<n+2; ++r)
    delete[] floor[r];
  delete[] floor;

  return 0;
}
```

Solution to Exercise 89. 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

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

```
// Program: euler_prime.cpp
// Finds a,b,c within specified bounds such that
// the formula |an^2 + bn + c| produces the largest
// number of distinct consecutive primes, starting
// with n = 0

#include <iostream>
#include <cassert>

int main ()
{
  // by multiplying with -1, if necessary, we may assume c >= 0
  const int arange = 100;      // a in {-arange+1,...,arange-1}
  const int brange = 1000;     // b in {-brange+1,...,brange-1}
  const int crange = 10000;    // c in {2,...,crange-1}

  // first, compute all primes in the set {an^2 + bn + c} where a, b, c
  // run through their ranges and n is smaller than 100 (we're not
  // searching for longer runs of primes here)
  const int max_elem = arange * 10000 + brange * 100 + crange;

  // run Eratosthenes' sieve
  bool prime[max_elem];
  for (int i = 2; i < max_elem; ++i)
    prime[i] = true;
  for (int i = 2; i < max_elem; ++i)
    if (prime[i]) {
      // cross out all proper multiples of i
      for (int m = 2*i; m < max_elem; m += i)
        prime[m] = false;
    }

  // now search for the best a, b, c
  int best_run = 0;
  int best_a = arange;
  int best_b = brange;
  int best_c = crange;

  // array to keep track of primes we have already seen in a run
  bool seen_before[max_elem];
  for (int i = 2; i < max_elem; ++i)
    seen_before[i] = false;

  // loop over all candidate triples (a,b,c)
  for (int c = 2; c < crange; ++c) {
    if (!prime[c]) continue; // not even a prime for n = 0
    for (int a = -arange+1; a < arange; ++a)
      for (int b = -brange+1; b < brange; ++b) {
        // evaluate elem = an^2 + bn + c for n=0,1,...
        int n = 0;
        int elem = c;
        for (; n < 100; ++n) {
          const int abs_elem = elem < 0 ?  -elem: elem;
          if (abs_elem < 2 || !prime[abs_elem]) break; // not a prime
```

```
          if (seen_before[abs_elem]) break;             // repeated prime
          seen_before[abs_elem] = true;                 // new prime
          // update element
          elem += a * (2*n + 1) + b;
        }
        // now we have seen a run of n primes (for 0,...,n-1)
        if (n > best_run) {
          best_run = n;
          best_a = a;
          best_b = b;
          best_c = c;
        }
        // remove the "seen_before" markers for the next run
        elem = c;
        for (int k=0; k<n; ++k) {
          const int abs_elem = elem < 0 ?  -elem: elem;
          seen_before[abs_elem] = false;
          elem += a * (2*k + 1) + b;
        }
      }
  }

  std::cout << "Best a        = " << best_a << ".\n";
  std::cout << "Best b        = " << best_b << ".\n";
  std::cout << "Best c        = " << best_c << ".\n";
  std::cout << "Euler quality = " << best_run << ".\n";

  return 0;
}
```

Solution to Exercise 90.

```
// Prog: xbm.cpp
// includes an xbm file and outputs the xbm-file that corresponds
// to the image rotated by 90 degrees

#include<iostream>
#include<cassert>
#include "original.xbm"
#define width original_width
#define height original_height
#define bits original_bits

int main()
{
  // array for mapping from {0,...,15} to {'0',...,'f'}
  char hex[] = {'0', '1', '2', '3', '4', '5', '6', '7',
                '8', '9', 'a', 'b', 'c', 'd', 'e', 'f'};

  // output header for rotated image
  std::cout << "#define rotated_width " << height << "\n";
  std::cout << "#define rotated_height " << width << "\n";
  std::cout << "static unsigned char rotated_bits[] = {\n";

  // go through the pixels columnwise (from right to left),
  // and within each column, proceed from top to bottom
  const unsigned int columns = (width+7)/8; // bytes in row (original)
  const unsigned int rows = (height+7)/8;   // bytes in column (rotated)
  bool comma = false;                 // not before first byte
  for (int c=columns-1; c>=0; --c) {
    // go through columns from right to left
    for (int d=128; d>0; d/=2) {
      // for d = 2^j, we are in column c+j
```

```
      for (int r=0; r<rows; ++r) {
        // go through rows from top to bottom and build
        // up one byte from any 8 pixels (least significant
        // digit comes from pixel in first of these rows)
        int byte = 0;
        for (int i=7; i>=0; --i) {
          // we are in row 8r+i
          int pixel;
          if (8*r+i >= height)
            // non-existing row, fill up with zeros
            pixel = 0;
          else
            // get pixel in row 8r+i and column c+j
            pixel = (unsigned char)(bits[(8*r+i)*columns+c])/d%2;
          byte = 2*byte+pixel;
        }
        if (comma) std::cout << ", "; comma = true;
        std::cout << "0x" << hex[byte/16] << hex[byte%16];
      }
    }
    std::cout <<"\n";
  }
  std::cout << "}\n;";

  return 0;

}
```