2.6 Arrays and pointers

Reading into an array without making a "silly error" is beyond the ability of complete novices - by the time you get that right, you are no longer a complete novice.

Bjarne Stroustrup, C++ Style and Technique FAQ

This section introduces arrays as containers for sequences of objects of the same type, with random access to individual members of the sequence. An array is the most primitive but at the same time a very efficient container for storing, processing, and iterating over large amounts of data. You will also learn about pointers as explicit object addresses and about their close relationship with arrays. While the C++ standard library contains less primitive and generally better alternatives, the concepts *behind arrays and pointers are of fundamental importance.*

In Section 2.4 on control statements, we have learned about the concept of iteration. For example, we can now iterate over the sequence of numbers $1, 2, \ldots, n$ and perform some operations like adding up all the numbers, or identifying the prime numbers among them. Similarly, we can iterate over the odd numbers, the powers of two, etc.

In real applications, however, we often have to process (and in particular iterate over) sequences of *data*. For example, if you want to identify the movie theaters in town that show your desired movie tonight, you have to iterate over the sequence of movie theater repertoires. These repertoires must be stored somewhere, and there must be a way to inspect them in turn. In C++, we can deal with such tasks by using *arrays*.

2.6.1 Array types

An array of length n aggregates n objects of the *same* type T into a sequence. To access one of the aggregated objects (the *elements*), we use its *index* or *subscript* (position) in the sequence. All these length-n sequences form an array type whose value range corresponds to the mathematical type T^n. In the computer's main memory, an array occupies a contiguous part, with the elements stored side-by-side (see Figure 6).

Let us start by showing an array in action: *Eratosthenes' Sieve* is a fast method for computing all prime numbers smaller than a given number n, based on crossing out the numbers that are not prime. It works like this: you write down the sequence of numbers between 2 and $n-1$. Starting from 2, you always go to the next number not crossed out yet, report it as prime, and then cross out all its proper multiples.

Let's not dwell on the correctness of this method but go right to the implementation. If you think about it for a minute, the major question is this: how do we cross out numbers?

The following program uses an array type variable `crossed_out` for the list, where any value `crossed_out[i]` is of type `bool` and represents the (changing) information whether the number `i` has already been crossed out or not. Array indices always start from 0, so in order to get to index $n-1$, we need an array of length n. The program runs Eratosthenes' Sieve for $n = 1,000$.

```
1  // Program: eratosthenes.C
2  // Calculate prime numbers in {2,...,999} using
3  // Eratosthenes' sieve.
4
5  #include <iostream>
6
7  int main()
8  {
9    // definition and initialization: provides us with
10   // Booleans crossed_out[0],..., crossed_out[999]
11   bool crossed_out[1000];
12   for (unsigned int i = 0; i < 1000; ++i)
13     crossed_out[i] = false;
14
15   // computation and output
16   std::cout << "Prime numbers in {2,...,999}:\n";
17   for (unsigned int i = 2; i < 1000; ++i)
18     if (!crossed_out[i]) {
19       // i is prime
20       std::cout << i << " ";
21       // cross out all proper multiples of i
22       for (unsigned int m = 2*i; m < 1000; m += i)
23         crossed_out[m] = true;
24     }
25   std::cout << "\n";
26
27   return 0;
28 }
```

Program 13: *progs/eratosthenes.C*

Definition. An array variable (or simply array) a with $n > 0$ elements of *underlying type* T is defined through the following declaration.

T a[*expr*]

Here, *expr* must be a *constant expression* of integral type whose value is n. For example, literals like `1000`, or arithmetic expressions over literals (like `1+1`) are constant

expressions; there are other constant expressions, but all of them have the property that their value is known at compile time. This allows the compiler to figure out how much memory the array variable needs.

The type of *a* is "T[n]", but we put this in double quotes here (only to omit them later). The reason is that T[n] is not the official name: we can't write `int[5] a`, for example, to declare an array `a` of type `int[5]`.

The value range of T[n] is T^n, the set of all sequences $(t_1, t_2, \ldots, t_n)$ with all t_i being of type T. The underlying type T might for example be any fundamental type (like `int`, `bool`, or `double`), and in this case, the values of the n array elements remain uninitialized by the definition.

The fact that the array length must be known at compile time clearly limits the usefulness of array variables. For example, this limitation does not allow us to write a version of Eratosthenes' sieve in which the number n is read from the input. But we will shortly see how this restriction can be overcome—for the time being, let's simply live with it.

2.6.2 Initializing arrays

The definition of an array with underlying fundamental type does not initialize the values of the array elements. We can assign values to the elements afterwards (like we do it in Program 13), but we can also provide the values directly, as in the following declaration statement.

```
int a[5] = {4,3,5,2,1};
```

Since the number of array elements can be deduced from the length of the *initializer list*, we can also write

```
int a[] = {4,3,5,2,1};
```

The declaration `int a[]` without any initialization is invalid, though, since it does not fully determine the type of `a`. We say that `a` has *incomplete type* in this case.

2.6.3 Random access to elements

The most common and useful way of accessing and modifying the elements of an array is by *random access*. If *expr* is of integral type and has value i, the lvalue

a[*expr*]

is of the type underlying the array `a` and refers to the i-th element (counting from 0) of `a`. The number i is called the *index* or *subscript* of the element. If n is the length of `a`, the index i must satisfy $0 \leq i < n$. The operator `[]` is called the *subscript operator*.

The somewhat strange declaration format of an array, with no explicit type name appearing, is motivated by the subscript operator. Indeed, the declaration

T a[*expr*]

can be read as "a[*expr*] is of type T". In this sense, it is an indirect definition of `a`'s type.

> **Watch out!** The C++ language offers no functionality for accessing the length of an array (see Section 2.6.4 below for more on this). As the programmer, *you* must remember the length yourself, and *you* are responsible for making sure that a given array index i indeed satisfies $0 \leq i < n$, where n is the length of the array. Indices that are not in this range are called *out of bound*. Unless your compiler offers specific debugging facilities, the usage of out-of-bound indices in the subscript operator is *not* detected at runtime and leads to undefined behavior of the program.

We have already discussed the term random access in connection with the computer's main memory (Section 1.2.3); random access means that *every* array element can be accessed in the same uniform way, and with (almost) the same access time, no matter what its index is. Evaluating the expression `a[0]` is as fast as evaluating `a[10000]`. In contrast, the thick pile of pending invoices, bank transfers and various other papers on your desk does not support random access: the time to find an item is roughly proportional to its depth within the pile.

In fact, random access in an array directly reduces to random access in the computer's main memory, since an array always occupies a contiguous set of memory cells, see Figure 6.

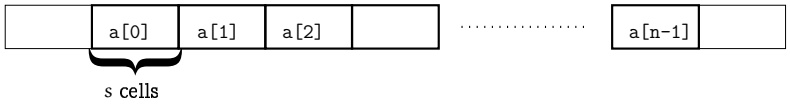


Figure 6: *An array occupies a contiguous part of the main memory. Every element in turn occupies s memory cells, where s is the memory required to store a single value of the underlying type T.*

To access the element of index i in the array `a`, a simple computation with addresses therefore suffices. If p is the address (position) where the first element of `a` "starts", and s is the number of memory cells that a single value of the underlying type T occupies, then the element of index i starts at the memory cell whose address is $p+si$, see Figure 7.

2.6.4 Arrays are not self-describing

Array types are exceptional in C++. The following code fragment illustrates this:

```
int a[5] = {4,3,5,2,1}; // array of type int[5]
int b[5];
b = a;                  // error: we cannot assign to an array
```

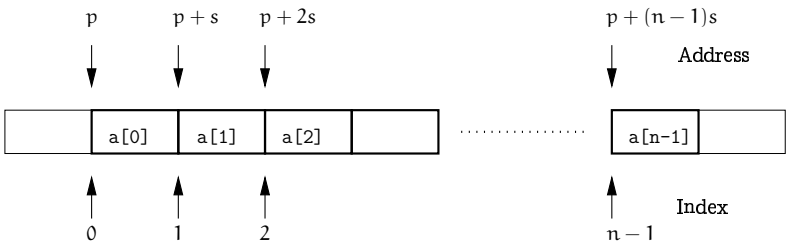


Figure 7: *The array element of index* i *starts at the address* $p + si$.

We also cannot initialize an array from another array. Why is this? Arrays are a dead hand from the programming language C, and the design of arrays in C is (from today's point of view) quite primitive. The main weakness is that *the number of elements of an array is not represented in the array's value*. C was designed to compete with machine language in efficiency, and this didn't leave room for luxury. An array variable is merely represented by its address, defined as the address of the memory cell where the first element (the one of index 0) starts. We also say that an array is not *self-describing*: it does not "know" its own length.

Now we get to the assignment issue: there is no way of automatically copying all elements of the array `a` into `b`, without knowing how many elements these are. But we *can* of course do this manually via a simple loop, since we know the array lengths from the declarations; early C programmers were not yet spoiled enough to complain about such minor inconveniences. (On the other hand, this leaves ample room for bugs.)

When C++ was developed much later, one design goal was to have C as a subset. As a consequence, arrays are still around in C++.

2.6.5 Iteration over a container

Let's take a step back, forget about the technicalities of arrays for a moment, and go for a bigger picture.

We have already indicated in the introduction to this section that the process of iterating over a sequence of data is ubiquitous. Typically, the data are stored in some *container*, and we need to perform a certain operation for all elements in the container. In general, a container is an object that can store other objects (its elements), and that offers some ways of accessing these elements. The only "hard" requirement here is that a container must offer the possibility of iterating over all its elements. In this informal sense, an array is indeed a container, since the random access functionality can be used to iterate over the elements.

Iteration by random access. Let's get back to arrays. Iterating over an array of length n can be done by random access like in lines 12–13 of Program 13. We have seen that the random access functionality of arrays is internally based on address arithmetic. During the iteration, the following sequence of addresses is computed: $p, p + s, p + 2s, \ldots, p + (n - 1)s$, where p and s have the usual meanings.

This requires one multiplication and one addition for any address except the first. But if you think about it, the multiplication only comes in because we compute each address from scratch, independently from the previous ones. In fact, the same set of addresses could more efficiently and more naturally be computed by starting with p and repeatedly adding s ("going to the next element").

Using random access, we can *simulate* array iteration, but we are missing the operation of "going to the next element"; only this operation makes iteration over a container natural and efficient. The following analogy illustrates the point: you *can* of course read a book by starting with page 1, then closing the book, opening it again on pages $2 - 3$, closing it, opening it on pages $4 - 5$, etc. But unless you're somewhat eccentric, you probably prefer to just turn the pages in between.

Iteration by pointers. Arrays offer natural and efficient iteration through *pointers*. Pointer values can be thought of as actual addresses, and they allow operations like "adding s" in order to go to the next element in the array. Here is how we could equivalently write the iteration in lines 12–13 of Program 13 with pointers.

```
bool* begin = crossed_out;       // pointer to first element
bool* end = crossed_out + 1000; // past-the-end pointer
// in the loop, pointer p successively points to all elements
for (bool* p = begin; p != end; ++p)
  *p = false; // *p is the element pointed to by p
```

Admittedly, this looks more complicated at first sight than the random access version, but we'll explain what's going on in detail in the next sections. In terms of Figure 7, we have replaced iteration by index with iteration by address.

2.6.6 Pointer types and functionality

For any type T the corresponding *pointer type* is

$T*$

We call T the underlying type of $T*$. An expression of type $T*$ is called a *pointer* (to T).

The value of a pointer to T is the address of an object of type T. We call this the object *pointed to* by the pointer.

We can visualize a pointer p as an arrow pointing to a cell in the computer's main memory—the cell where the object pointed to by p starts, see Figure 8.

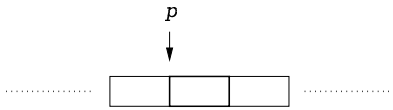


Figure 8: *A pointer to T represents the address of an object of type T in the computer's main memory.*

Initialization, the assignment operator =, and the comparison operators == and != are defined for any pointer type *T**. The latter simply test whether the addresses in question are the same or not.

Initialization and assignment copy the value (as usual), which in this case means to copy an address; thus, if `j` points to some object, the assignment `i = j` has the effect that `i` now also points to this object. The object itself is not copied. We remark that pointer initialization and assignment require the types of both operands to be exactly the same—implicit conversions don't work. If you think about it, this is clear. Imagine that the variable `i` is of type `int*`, and that you could write

```
double* j = i
```

Since `double` objects usually require more memory cells than `int` objects, `j` would now be a pointer to a `double` object that includes memory cells originally not belonging to `i`. This can hardly be called a "conversion". In fact, since we only copy an address, there cannot be any physical conversion of the stored value, even if the memory requirements of the two types happen to be the same.

The address operator. We can obtain a pointer to any given object by applying the unary *address operator* to any lvalue that refers to the object. If the lvalue is of type `T`, then the result is an rvalue of type `T*`. The syntax of an address operator call is

&lvalue

In the following code fragment we use the address operator to initialize a variable `iptr` of type `int*` with the address of an object of type `int` named `i`.

```
int i = 5;
int* iptr = &i; // iptr initialized with the address of i
```

The dereference operator. From a pointer, we can get back to the object pointed to through *dereferencing* or *indirection*. The unary *dereference operator* * applied to an rvalue of pointer type yields an lvalue referring to the object pointed to. If the rvalue is of type *T**, then the result is of type *T*. The syntax of a dereference operator call is

**rvalue*

Following up on our previous code fragment, we can therefore write

```
int i = 5;
int* iptr = &i; // iptr initialized with the address of i
int j = *iptr;  // j == 5
```

The naming scheme of pointer types is motivated by the dereference operator. The declaration

T p*

can also be read (and in fact legally be written; we don't do this, though) as

*T *p*

The second version indirectly defines the type of `p` by saying that **p* is of type *T*. This is the same kind of indirect definition that we already know from array declarations.

Figure 9 illustrates address and dereference operator.

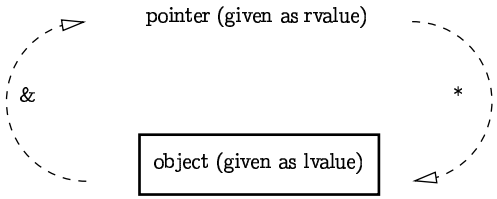


Figure 9: *The address operator (left) and its inverse, the dereference operator (right)*

The null pointer. For any pointer type there is a value distinguishable from any other pointer value. This value is called the *null pointer value*. The integer value `0` can be converted to any pointer type. The value after conversion is the null pointer value. In the declaration `int* iptr = 0`, for example, the variable `iptr` gets initialized with the null pointer value. We also say that `iptr` is a *null pointer*. The null pointer value must not be dereferenced, since it does not correspond to any existing address.

Using the null pointer value is the safe way of indicating that there is no object (yet) to point to. The alternative of leaving the pointer uninitialized is bad: there is no way of testing whether a pointer that is not a null pointer holds the address of a legitimate

object, or whether it holds some "random" address resulting from leaving the pointer uninitialized.

In the latter case, dereferencing the pointer usually crashes the program. Consider this code:

```
int* iptr;     // uninitialized pointer
int j = *iptr; // trouble!
```

After its declaration, the pointer `iptr` has undefined value, which in practice means that it may correspond to an arbitrary address in memory; dereferencing it means to access the memory content at this address. In general, this address will not belong to the part of memory to which the program has access; the operating system will then deny access to it and terminate the program with a *segmentation fault*.

2.6.7 Array-to-pointer conversion

Any array of type $T[n]$ can implicitly be converted to type $T*$. The resulting value is the address of the first element of the array. For example, we can write

```
int a[5];
int* begin = a; // begin points to a[0]
```

The declaration

```
int* begin = &a[0]; // address of the first element
```

is equivalent as far as the resulting value of `begin` is concerned, but there is a subtle difference: the latter declaration evaluates `a[0]`, while the former does not.

The pointer-style replacement code for the loop in lines 12–12 of Program 13 that we have presented at the end of Section 2.6.5 makes use of array-to-pointer conversion in the first line:

```
bool* begin = crossed_out;       // pointer to first element
```

The array-to-pointer conversion is purely conceptual; on the machine side, nothing happens. For this, we recall from our earlier discussion in Section 2.6.4 that in C and therefore also in C++, an array "is" simply the address of its first element.

Array-to-pointer conversion automatically takes place when an array appears in an expression.[22] Bjarne Stroustrup, the designer of C++, illustrates this by saying that *the name of an array converts to a pointer to its first element at the slightest provocation*. In still other words, there are no operations on arrays: everything that we conceptually do with an array is in reality done with a pointer; this in particular applies to the random access operation, see the paragraph called "Pointer subscripting, or the truth about random access" in the next section.

[22] A notable exception is the case where an array appears as the left operand of an assignment. This does not trigger array-to-pointer conversion but an error message saying that arrays can't be assigned to.

2.6.8 Pointer arithmetic

In order to understand why the code fragment

```
bool* begin = crossed_out;       // pointer to first element
bool* end = crossed_out + 1000; // past-the-end pointer
// in the loop, pointer p successively points to all elements
for (bool* p = begin; p != end; ++p)
  *p = false; // *p is the element pointed to by p
```

indeed sets all elements of the array `crossed_out` to *false*, we have to understand *pointer arithmetic*, the art of computing with addresses. We deliberately call this an "art", since pointer arithmetic comes with a lot of pitfalls, but without a safety net. On the other hand, the authors feel that there is also a certain beauty in the minimalism of pointer arithmetic. It's like driving an oldtimer: it's loud, it's difficult to steer, seats are uncomfortable, and there's no heating. But the heck with it! The oldtimer looks so much better than a modern car. Nevertheless, after driving the oldtimer for a while, it will probably turn out that beauty is not enough, and that safety and usability are more important factors in the long run.

Adding integers to pointers. The binary addition operators +, - are defined for left operands of any pointer type $T*$ and right operands of any integral type. Recall that if an array is provided as the left operand, it will implicitly be converted to a pointer using array-to-pointer conversion.

For the behavior of + to be defined, there must be an array of some length n, such that the left operand *ptr* is a pointer to the element of some index $k, 0 \leq k \leq n$, in the array. The case $k = n$ is allowed and corresponds to the situation where *ptr* is a pointer one past the last element of the array (we call this a *past-the-end pointer*; note that such a pointer must not be dereferenced).

If the second operand *expr* has some value i such that $0 \leq k + i \leq n$, then

ptr + *expr*

is a pointer to the $(k + i)$-th element of the same array. Informally, we get a pointer that has been moved "i elements to the right" (which actually means to the left if i is negative).

Again, if $k + i = n$, we get a past-the-end pointer. Values of i such that $k + i$ is not between 0 and n lead to undefined behavior.

Let us repeat the point that we have made before in connection with random access in Section 2.6.3: by default, there are absolutely no checks that the above requirements indeed hold, and it is entirely your responsibility to make sure that this is the case. Failure to do so will result in program crashes, strange behavior of the program, or (probably the worst scenario) seemingly normal behavior, but with the potential of turning into strange behavior at any time, or on any other machine.

Therefore, let us summarize the requirements once more:

- *ptr* must point to the element of index k in some array of length n, where $0 \leq k \leq n$, and
- *expr* must have some value i such that $0 \leq k + i \leq n$.

Binary subtraction is similar. If *expr* has value i such that $0 \leq k - i \leq n$, then

ptr − *expr*

yields a pointer to the array element of index $k - i$.

The assignment versions += and -= of the two operators, and the unary increment and decrement operators ++ and -- can be used with left operands of pointer type as well, with the usual meaning. Since precedences and associativities are tied to the operator symbols, they are as in Table 1 on page 42.

Now we can understand the second line of the above code fragment:

```
bool* end = crossed_out + 1000; // pointer after last element
```

First, the array `crossed_out` is converted to a pointer to its first element (the one of index 0). Since the array has 1,000 elements, adding the integer 1,000 yields a past-the-end pointer `end` for the array. The subsequent loop

```
for (bool* p = begin; p != end; ++p)
  *p = false; // *p is the element pointed to by p
```

is clear now as well: starting with a pointer `p` to the first element (`p = begin`), the element pointed to is set to *false* (`*p = false`). Then we increment `p` so that it points to the next element (`++p`). We repeat this as long as `p` is different from the past-the-end pointer named `end`.

Pointer comparison. We have already discussed the relational operators == and != that simply test whether the two pointers in question point to the same object. But we can also compare two pointers using the operators <, <=, >, and >=. Again, precedences and associativities of all relational operators are as in Table 2 on page 63.

For the result to be specified, there must be an array of some length n, such that the left operand *ptr1* is a pointer to the element of some index $k_1, 0 \leq k_1 \leq n$ in the array, and the second operand *ptr2* is a pointer to the element of some index $k_2, 0 \leq k_2 \leq n$ in the same array. Again, $k_1 = n$ and $k_2 = n$ are allowed and correspond to the past-the-end case.

Given this, the result of the pointer comparison is determined by the integer comparison of k_1 and k_2. In other words (and quite intuitively), the pointer to the element that comes first in the array is the smaller one.

In our code fragment, the comparison `p < end` therefore yields *true* as long as `p` is not a past-the-end pointer; equivalently, as long as `p` points to an actual array element. The loop therefore does what we want: it sets all array elements to *false*.

Comparing two pointers that do not meet the above requirements leads to unspecified results in the four operators <, <=, >, and >=.

Pointer subtraction. There is one more arithmetic operation on pointers. Assume that *ptr1* is a pointer to the element of some index $k_1, 0 \leq k_1 \leq n$ in some array of length n, and the second operand *ptr2* is a pointer to the element of some index $k_2, 0 \leq k_2 \leq n$ in the same array (past-the-end pointers allowed). Then the result of the *pointer subtraction*

ptr1 − *ptr2*

is the integer $k_1 - k_2$. Thus, pointer subtraction tells us "how far apart" the two array elements are. The behavior of pointer subtraction is undefined if *ptr1* and *ptr2* are not pointers to elements in (or past-the-end pointers of) the same array.

Pointer subtraction (which employs the binary subtraction operator, see Table 1 on page 42 for its specifics) does not occur in the code fragment from the beginning of this section. A typical use is to determine the number of elements in an array that is given by a pointer to its first element and a past-the-end pointer.

Pointer subscripting, or the truth about random access. In reality, the subscript operator [] as introduced in Section 2.6.3 does not operate on arrays, but on pointers. Invoking this operator on an array constructs an expression and therefore triggers an array-to-pointer conversion.

Given a pointer *ptr* and an expression *expr* of integral type, the expression

ptr[*expr*]

is equivalent (also in its requirements on *ptr* and *expr*) to

$*(ptr + expr)$

If *expr* has value i, the latter expression yields the array element i places to the right of the one pointed to by *ptr*. In particular, if *ptr* results from an array-to-pointer conversion, this agrees with the semantics of random access for arrays as introduced in Section 2.6.3.

Table 4 summarizes the new pointer-specific binary operators.

Description	Operator	Arity	Prec.	Assoc.
subscript	[]	2	17	left
dereference	*	1	16	right
address	&	1	16	right

Table 4: *Precedences and associativities of pointer operators. The subscript operator expects rvalues as operands and returns an lvalue. The dereference operator expects an rvalue and returns an lvalue, while the address operator expects an lvalue and returns an rvalue.*

What have we gained with pointers? So far it seems that the only use of pointers is to make iteration through an array a little more efficient than iteration by index. But unless we are in the realm of extremely time-critical loops, the savings are marginal. For the sake of readability, we therefore often still use iteration by index. So what is the real justification for the pointer concept?

There are actually two justifications, and one of them will be discussed right away in the next section: pointers are indispensable for getting "practical" arrays with length not known at compile time.

The second justification is not yet around the corner, so we will only briefly touch it here. Arrays are by far not the only containers for sets of data. When we implement data processing algorithms, we should therefore make sure that they work *not* only for arrays.

For example, finding a container element with a given property (movie theater that plays your favorite movie) should be possible for *any* containers that offers the functionality of iterating over its elements. The only uniformity we need is in the iteration process itself.

Any data-processing algorithm of the C++ standard library (we will see some of them later) works in this way: it expects the underlying container to offer *iterators* conforming to some well-defined iterator concept. The specifics of the container itself are irrelevant for the algorithm.

Here is where pointers come in: they are the iterators offered by arrays. Therefore, even if we don't use pointers in our own code, we have to know about them in order to be able to apply standard library algorithms to arrays.

2.6.9 Dynamic memory allocation

Let us go back to Program 13 now. Its main drawback is that the number n is hardwired as $1,000$ in this program, just because the length of an array has to be known at compile time.

At least in this respect, arrays are nothing special, though. All types that we have met earlier (`int`, `unsigned int`, and `bool`) have the property that a single object of the type occupies a fixed amount of memory known to the compiler (for example, 32 bits for an `int` object on many platforms). With arrays, an obvious need arises to circumvent this restriction.

In C++, arrays whose length is determined at runtime can be obtained through *dynamic memory allocation*. Through such an allocation, we create an object with *dynamic* storage duration.

Objects that we have seen so far were all tied to variables, in which case memory gets assigned to them (and is freed again) at predetermined points during program execution (automatic and static storage duration, Section 2.4.3). Objects of dynamic storage duration are not tied to variables, and they may "start to live" (get memory assigned to them) and "die" (get their memory freed) at *any* point during program execution. The programmer can determine these points via `new` and `delete` expressions.

The program has some (typically quite large) region of the computer's main memory available to store dynamically allocated objects. This region is called the *heap*. It is initially unused, but when an object is dynamically allocated, it is being stored on the heap, so that the memory actually used by the program grows.

Here is how this works for Eratosthenes' Sieve. Remember that we want the list of prime numbers between 2 and $n-1$. The following variant reads the number n from standard input and dynamically allocates an array of length n. The remainder of the program is as before, except that we explicitly have to free the dynamically allocated storage in the end.

```
1  // Program: eratosthenes2.C
2  // Calculate prime numbers in {2,...,n-1} using
3  // Eratosthenes' sieve.
4
5  #include <iostream>
6
7  int main()
8  {
9    // input
10   std::cout << "Compute prime numbers in {2,...,n-1} for n =? ";
11   unsigned int n;
12   std::cin >> n;
13
14   // definition and initialization: provides us with
15   // Booleans crossed_out[0],..., crossed_out[n-1]
16   bool* crossed_out = new bool[n];         // dynamic allocation
17   for (unsigned int i = 0; i < n; ++i)
18     crossed_out[i] = false;
19
20   // computation and output
21   std::cout << "Prime numbers in {2,...," << n-1 << "}:\n";
22   for (unsigned int i = 2; i < n; ++i)
23     if (!crossed_out[i]) {
24       // i is prime
25       std::cout << i << " ";
26       // cross out all proper multiples of i
27       for (unsigned int m = 2*i; m < n; m += i)
28         crossed_out[m] = true;
29     }
30   std::cout << "\n";
31
32   delete[] crossed_out;                    // free dynamic memory
33
34   return 0;
35 }
```

Program 14: *progs/eratosthenes2.C*

Note that the variable `crossed_out` is now a pointer rather than an array; after the `new` declaration, it points to the first element of a dynamically allocated array of length `n`.

The new expression. For any type T, a `new` expression can come in any of the following three variants.

```
new T
new T(...)
new T[expr]
```

In all cases, the expression returns an rvalue of type T*. Its value is the address of an object of type T that has been dynamically allocated on the heap. The object itself is anonymous, but we usually store the resulting address under a variable name. In Program 14, we call it `crossed_out`.

In the first and second variant, the effect of the `new` expression is to dynamically allocate a *single* object of type `T` on the heap. Variant 1 leaves the object uninitialized if T is a fundamental type, while variant 2 initializes the new object with whatever appears in parentheses. For example, the following declarations initialize the variables `i` and `j`, both of type `int*`, with the addresses of two new objects of type `int`.

```
int* i = new int;      // *i is undefined
int* j = new int(6);   // *j is 6
```

Right now, if we wanted two such objects of type `int`, we'd rather use variables with automatic storage duration and write

```
int i;                 // i is undefined
int j = 6;             // j is 6
```

More interesting for us is the third variant. If *expr* has integer value $n \geq 0$, the effect of the `new` expression is to dynamically allocate an array of length n with underlying type T on the heap. The return value is the address of the first element. This is what we see in line 16 of Program 14.

As usual, the n array elements remain uninitialized if T is a fundamental type.

The delete expression. Dynamically allocated memory that is no longer needed should be freed. In C++, the programmer decides at which point this is the case. [23] Dynamic

[23] There are programming languages (Java, for example) that automatically detect and free unused memory on the heap. This automatic process is called *garbage collection*. It is generally more user-friendly than the manual deletion process in C++, but requires a more sophisticated implementation. In any case, you are free to implement garbage collection also in C++.

storage duration implies that dynamically allocated objects live until the program terminates, unless they are explicitly freed. Dynamically allocated memory is more flexible than static memory, but in return it also involves some administrative effort.

The `delete` expressions take care of freeing memory. They come in two variants.

```
delete expr
delete[] expr
```

In both variants, *expr* may be a null pointer, in which case the `delete` expression has no effect.

Otherwise, in the first variant, *expr* must be a pointer to a single object that has previously been dynamically allocated with the first or second variant of the `new` expression. The effect is to make the corresponding memory available again for subsequent dynamic allocations on the heap.

For example, at a point in the program where the two `int` objects dynamically allocated through

```
int* i = new int;      // *i is undefined
int* j = new int(6);   // *j is 6
```

are no longer needed, we would write

```
delete j;
delete i;
```

The order of deletion does not matter here, but many programmers consider it logical to `delete` pointers in the inverse order of dynamic allocation: If you need to undo two steps, you first undo the second step.

In the second variant of the `delete` expression, *expr* must be a pointer to the first element of an array that has previously been dynamically allocated with the third variant of the `new` expression. The whole memory occupied by the array is put back on the heap for reuse.[24] This happens in line 32 of Program 14.

If the plain `delete` is applied to a non-null pointer that does not point to a dynamically allocated single object, the behavior is undefined. The same is true if one tries to `delete[]` an array where there is only a single object. As always with pointers, the C++ language does not offer any means of detecting such errors.

Memory leaks. Although all memory allocated by a program is automatically freed when the program terminates normally, it is very bad practice to rely on this fact for freeing dynamically allocated memory. If a program does not explicitly free all dynamically allocated memory it is said to have a *memory leak*. Such leaks are often a sign of bad coding. They usually have no immediate consequences, but without freeing unused

[24] This implies that the length of a dynamically allocated array *is* actually stored somewhere with the heap; still, we can't access this length from the program.

storage, a program running for a long time (think of operating system routines) may at some point simply exhaust the available heap storage.

Therefore, we have the following guideline.

Dynamic Storage Guideline:
new and delete expressions should always come in matching pairs.

2.6.10 Arrays of characters

Sequences of characters enclosed in double quotes like in

```
std::cout << "Prime numbers in {2,...,999}:\n";
```

are called *string literals*.[25]

So far we have used string literals only within output expressions, but we can work with them in other contexts as well. Most notably, a string literal can be used to initialize an array of *characters*. Characters are the building blocks of text as we know it. In C++, they are modeled by the fundamental type `char` that we briefly discuss next.

The type char. The fundamental type `char` represents characters. Characters include the *letters* `a` through `z` (along with their capital versions `A` through `Z`), the *digits* `0` through `9`, as well as numerous other *special characters* like `%` or `$`. The line

```
char c = 'a';
```

defines a variable `c` of type `char` and value `'a'`, representing the letter `a`. The expression `'a'` is a literal of type `char`. The quotes around the actual character symbol are necessary in order to distinguish the literal `'a'` from the identifier `a`.

Formally, the type `char` is an integral type: it has the same operators as the types `int` or `unsigned int`, and the C++ standard even postulates a promotion from `char` to `int` or `unsigned int`. It is *not* specified, though, to which integer the character `'a'`, say, will be promoted. Under the widely used `ASCII` code (American Standard Code for Information Interchange), it is the integer 97.

This setting may not seem very useful, and indeed it makes little sense to divide one character by another. On the other hand, we can for example print the alphabet through one simple loop (assuming `ASCII` encoding). Execution of the `for`-loop

```
for (char c = 'a'; c <= 'z'; ++c)
  std::cout << c;
```

writes the character sequence

```
abcdefghijklmnopqrstuvwxyz
```

[25] Unlike all other literals, string literals are lvalues, but the effect of trying to modify them is undefined; luckily, we won't need these "interesting" facts.

to standard output. Given this, you may think that the line

```
std::cout << 'a' + 1;
```

prints `'b'`, but it doesn't. Since the operands of the composite expression `'a'+1` are of different types, the left operand of type `char` will automatically be promoted to the more general type `int` of the right operand. Therefore, the type of the expression `'a'+1` is `int`, and its value is 98 (assuming `ASCII` encoding); and that's what gets printed. If you want `'b'` to be printed, you must use the explicit conversion `char('a'+1)`.

The category of special characters also includes *control characters* that *do* something when printed. These are written with a leading backslash, and the most important control character for us is `'\n'`, which causes a line break.

On most platforms, a `char` value occupies 8 bits of memory; whether the value range correspond to the set of integers $\{-128,\ldots,127\}$ (the signed case) or the set $\{0,\ldots,255\}$ (the unsigned case) is implementation defined. Since all ASCII characters have integer values in $\{0,\ldots,127\}$, they can be represented in both cases.

From characters to text. A text is simply a sequence of characters and can be modeled in C++ through an array with underlying type `char`. For example, the declaration

```
char text[] = {'b', 'o', 'o', 'l'}
```

defines an array of length 4 that represents the text *bool*.

Alternatively (and more conveniently), we can write

```
char text[] = "bool"
```

This, however, is *not* equivalent to the former declaration. When an array of characters is initialized with a string literal, the terminating *zero character* `'\0'` (of integer value 0) is automatically appended to the array. This character does not correspond to any printable character. After the latter declaration, the array `text` therefore has length 5. The first four elements are `'b'`, `'o'`, `'o'`, and `'l'`, and the fifth element is the zero character `'\0'`.

We call such an array *zero-terminated*. Unlike normal arrays, zero-terminated arrays "know" their length. To get this length, we simply have to iterate over the array and count the number of elements before the terminating `'\0'`.

Here is an application of (arrays of) characters. *String matching* is the problem of finding the first or all occurrences of a given search string (usually short) in a given text (usually long).

The obvious solution is the following: assuming that the search string has length m, we compare it characterwise with the elements $1, 2, ..., m$ of the text. If a mismatch is found for some element, we stop and next compare the search string with the elements $2, 3, \ldots, m+1$ of the text, and so on. Sets of m consecutive elements $i, i+1, \ldots, i+m-1$ in the text are called a *window*.

This algorithm is fast as long as the search string is short, but it may become inefficient for long search strings (see Exercise 62). There is a more sophisticated algorithm (the *Knuth-Morris-Pratt algorithm*) that is always fast.

The following Program 15 implements the obvious algorithm. It maintains two arrays of characters, one for the search string, and one for the current window. We impose a cyclic order on the window (the first element directly follows the last one); this makes it easy to shift the window one place, by simply replacing element i of the text with element i + m (and at the same time advancing the logical first position of the window by one).

```
1  // Program: string_matching.C
2  // find the first occurrence of a fixed string within the
3  // input text, and output the text so far
4
5  #include<iostream>
6
7  int main ()
8  {
9    // search string
10   char s[] = "bool";
11
12   // determine search string length m
13   unsigned int m = 0;
14   for (char* p = s; *p != '\0'; ++p) ++m;
15
16   // cyclic text window of size m
17   char* t = new char[m];
18
19   unsigned int w = 0; // number of characters read so far
20   unsigned int i = 0; // index where t logically starts
21
22   // find pattern in the text being read from std::cin
23   std::cin >> std::noskipws; // don't skip whitespaces!
24
25   for (unsigned int j = 0; j < m;)
26     // compare search string with window at j-th element
27     if (w < m || s[j] != t[(i+j)%m])
28       // input text still too short, or mismatch:
29       // advance window by replacing first character
30       if (std::cin >> t[i]) {
31         std::cout << t[i];
32         ++w;          // one more character read
33         j = 0;        // restart with first characters
34         i = (i+1)%m;  // of string and window
35       } else break;   // no more characters in the input
36     else ++j; // match: go to next character
37
38   std::cout << "\n";
```



```
39   delete[] t;
40   return 0;
41 }
```

Program 15: *progs/string_matching.C*

When we apply the program to the text of the file `eratosthenes.C`, the program outputs Program 13 up to the first occurrence of the string `"bool"`:

```
// Program: eratosthenes.C
// Calculate prime numbers in {2,...,999} using
// Eratosthenes' sieve.

#include <iostream>

int main()
{
  // definition and initialization: provides us with
  // Booleans crossed_out[0],..., crossed_out[999]
  bool
```

A few comments need to be made with respect to the handling of standard input here. The program reads the text character by character from `std::cin`, until this stream becomes "empty". To test this, we use the fact that stream values can implicitly be converted to `bool`, with the result being *true* as long as there was no attempt at reading past the end of the stream. Since the value of `std::cin >> t[i]` is the stream *after* removal of one character, the conversion to `bool` exactly tells us whether there still was a character in the stream, or not.

Most conveniently, the program is run by redirecting standard input to a file containing the text. In this case, the stream `std::cin` will become empty exactly at the end of the file. The line

```
std::cin >> std::noskipws; // don't skip whitespaces!
```

is necessary to tell the stream that *whitespaces* (blanks, newlines, etc.) should not be ignored (by default, they are). This allows us to search for strings that contain whitespaces, and it allows us to output the text (up to the first occurrence of the search string) in its original layout.

2.6.11 Multidimensional arrays

In C++, we can have arrays of arrays. For example, the declaration

```
int a[2][3]
```

declares `a` to be an array of length 2 whose elements are arrays of length 3 with underlying type `int`. We also say that `a` is a *multidimensional array* (in this case of dimensions 2 and 3). The type of `a` is "`int[2][3]`", and the underlying type is `int[3]`. In general, the declaration

T a[*expr1*]...[*exprk*]

defines an array a of length n_1 (value of *expr1*) whose elements are arrays of length n_2 (value of *expr2*) whose elements are... you get the picture. The values $n_1, \ldots, n_k$ are called the *dimensions* of the array, and the expressions *expr1*,...,*exprk* must be constant expressions of integral type and positive value.

Random access in multidimensional arrays works as expected: `a[i]` is the element of index i, and this element is an array itself. Consequently, `a[i][j]` is the element of index j in the array `a[i]`, and so on.

Although we usually think of multidimensional arrays as tables or matrices, the memory layout is "flat" like for one-dimensional arrays. For example, the twodimensional array declared through `int a[2][3]` occupies a contiguous part of the memory, with space for $6 = 2 \times 3$ objects of type `int`, see Figure 10.

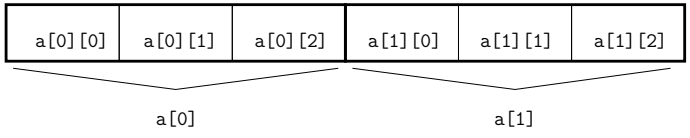


Figure 10: *Memory layout of a twodimensional array*

Multidimensional arrays can be initialized in a way similar to onedimensional arrays; the value for the *first* (and *only* the first) dimension may be omitted:

```
int a[][3] = { {2,4,6}, {1,3,5} };
```

This defines an array of type `int[2][3]` where `{2,4,6}` is used to initialize the element `a[0]`, and `{1,3,5}` is used for `a[1]`.

Dynamic allocation of multidimensional arrays. The required dimensions of a multidimensional array may not be known at compile time in which case dynamic allocation is called for. Let us start with the case where all dimensions but the first are known at compile time. If *expr* has value $n \geq 0$, a pointer to a dynamically allocated array of length n with underlying type $T[n_2]...[n_k]$ is obtained from a `new` expression

`new` *T*[*expr*][*expr2*]...[*exprk*]

where *expri* has value n_i, $i = 1, \ldots, k$. All dimensions but the first must be constant expressions. If you think about it for a minute, this is not surprising. For example, in order to generate machine language code for random access operations on the dynamically allocated array, the compiler must know how many memory cells a single element of the underlying type $T[n_2]...[n_k]$ occupies (see Section 2.6.3). But this is only possible if the values $n_2,\ldots,n_k$ are known at compile time.

Pointers to arrays. If we want to use the above `new` expression to initialize a pointer variable (with the address of the first element of the multidimensional array), we need the type "pointer to $T[n_2]...[n_k]$". As you may suspect, we informally call this type "$T[n_2]...[n_k]$*", but we can't write it like that in C++, since $T[n_2]...[n_k]$ is not a type name. Again, we have to resort to an indirect definition of the desired pointer variable p, as in the following code fragment.

```
int n = 2;
int (*p)[3] = new int[n][3];  // type of *p: int[3] <=> p: int[3]*
```

The parentheses are necessary here, since `int *p[3]` (which is the same as `int* p[3]`) declares p to be an array of pointers to `int` (see also next paragraph). C++ syntax is bittersweet.

Arrays of pointers. If you're asking for a multidimensional array with non-constant dimensions among $n_2, \ldots, n_k$, the official answer is: there is none. But under the counter, you can buy a very good imitation.

One first solution that suggests itself when you reconsider the flat memory layout of multidimensional arrays is this: you dynamically allocate a onedimensional array of length $n = n_1 \times n_2 \times \cdots \times n_k$ and artificially partition it into subarrays by doing some juggling with indices.

Let us discuss the twodimensional case only to avoid lengthy formulae. A twodimensional array with dimensions n and m can be simulated by a onedimensional array of length nm. The element with logical indices $i \in \{0, 1, \ldots, n-1\}$ and $j \in \{0, 1, \ldots, m-1\}$ appears at index $mi + j$ in the onedimensional array. Vice versa, the element of index ℓ in the onedimensional array has logical indices $i = \ell \operatorname{div} m$ and $j = \ell \bmod m$. This works because the function

$$(i, j) \mapsto mi + j$$

bijectively maps the set of logical indices (i, j) to the set of numbers $\{0, 1, \ldots, nm-1\}$. Intuitively, this mapping flattens the imaginary table of n rows and m columns by simply putting one row after another. As you can see from Figure 10, this is exactly what the compiler is implicitly doing for multidimensional arrays with *constant* dimensions $n_1, \ldots, n_k$.

Doing it explicitly for non-constant dimensions is only a workaround, though, since we lose the intuitive notation `a[i][j]`; moreover, this workaround becomes even more cumbersome with higherdimensional arrays.

A better solution that keeps the notation `a[i][j]` and that smoothly extends to higher dimensions is the following (again, we only discuss the case of a twodimensional array with dimensions n and m): you first dynamically allocate one array of n *pointers*, and then you let every single pointer point to the first element of an individual, dynamically allocated array of length m. The following code fragment demonstrates this.

```
// a points to the first element of an array of n pointers to int
int** a = new int*[n];
for (int i = 0; i < n; ++i)
  // a[i] points to the first element of an array of m int's
  a[i] = new int[m];
```

The type `int**` is "pointer to pointer to `int`". `a[i]` is therefore a pointer to `int` (see the paragraph on pointer subscripting in Section 2.6.8), and `a[i][j]` is an lvalue of type `int`, just like in a "regular" twodimensional array.

The memory layout is different, though: Figure 10 is replaced by Figure 11. This means, the twodimensional array is patched up from a set of n onedimensional arrays, but these n arrays are not necessarily consecutively arranged in memory. In fact, the n arrays may even have different lengths. This is useful for example when you want to store a lower-triangular matrix; in this case, it suffices if the row of index i has length $i+1$.

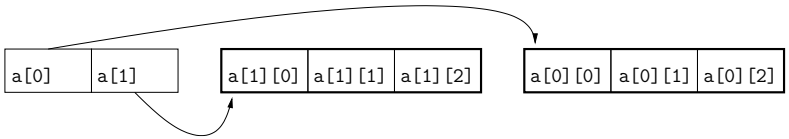


Figure 11: *Memory layout of a twodimensional array realized by an array of pointers*

Computing shortest paths. Let us conclude this section with an interesting application of (multidimensional) arrays. Imagine a rectangular factory floor, subdivided into square cells. Some of the cells are blocked with obstacles (these could for example be machines or cupboards, but let us abstractly call them "walls"). A robot is initially located at some cell S (the source), and the goal is to move the robot to some other cell T (the target). At any time, the robot can make one step from its current cell to any of the four adjacent cells, but for obvious reasons it may only use cells that are empty.

Given this setup, we want to find a shortest possible robot path from S to T (or find out that no such path exists). Here, the length of a robot path is the number of steps taken by the robot during its motion from S to T (the initial cell S does not count; in particular, it takes 0 steps to reach S from S). Figure 12 (left) shows an example with 8×12 cells.

In this example, a little thinking reveals that there are essentially two different possibilities for the robot to reach T: it can pass below the component of walls adjacent to S, or above. It turns out that passing above is faster, and a resulting shortest path (of length 21) is depicted in Figure 12 (right). Note that in general there is not a unique shortest path. In our example, the final right turn of the path could also have been made one or two cells further down.

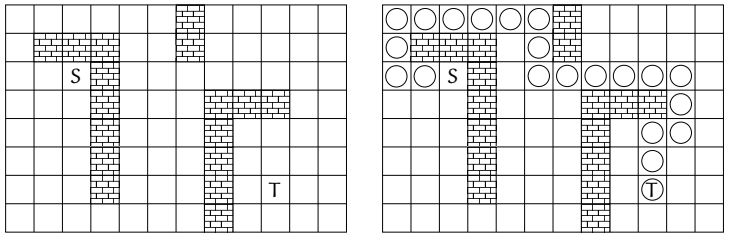


Figure 12: *Left: What is a shortest robot path from* S *to* T*? Right: This one!*

We want to write a program that finds a shortest robot path, given the dimensions n (number of rows) and m (number of columns) of the factory floor, the coordinates of source and target, and the walls. How can this be done? Before reading further, we encourage you to think about this problem for a while. Please note that the *brute-force approach* of trying all possible paths and selecting the shortest one is not an option, since the number of such paths is simply too large already for moderate floor dimensions. (Besides, how do you even generate all these paths?)

Here is an approach based on *dynamic programming*. This general technique is applicable to problems whose solutions can quickly be obtained from the solutions to smaller subproblems of the same structure. The art in dynamic programming is to find the "right" subproblems, and this may require a more or less far-reaching generalization of the original problem.

Once we have identified suitable subproblems, we solve all of them in turn, from the smaller to the larger ones, and memorize the solutions. That way, we have all the information that we need in order to quickly compute the solution to a given subproblem from the solutions of the (already solved) smaller subproblems.

In our case, we generalize the problem as follows: for *all* empty cells C on the floor, compute the *length* of a shortest path from S to C (where the value is ∞ if no such path exists). We claim that this also solves our original problem of computing a shortest path from S to T: Assume that the length of a shortest path from S to T is $\ell < \infty$ (otherwise we know right away that there is no path at all). We also say that T is *reachable* from S in ℓ steps.

Now if $T \neq S$, there must be a cell adjacent to T that is reachable from S in $\ell - 1$ steps, and adjacent to this a cell reachable in $\ell - 2$ steps etc. Following such a chain of cells until we get to S gives us a path of length ℓ which is shortest possible.

Let us rephrase the generalized problem: we want to label any empty cell C with a nonnegative integer (possibly ∞) that indicates the length of a shortest path from S to C. Here are the subproblems to which we plan to reduce this: for a given integer $i \geq 0$, label all the cells that are reachable from S in at most i steps. For $i = nm - 1$

(actually, for some smaller value), this labels all cells that are reachable from S at all, since a shortest path will never enter any cell twice.

Here is the reduction from larger to smaller subproblems: assume that we have already solved the subproblem for $i-1$, i.e. we have labeled all cells that are reachable from S within $i-1$ or less steps. In order to solve the subproblem for i, we still need to label the cells that are reachable in i steps (but not less). But this is simple, since these cells are exactly the unlabeled ones adjacent to cells with label $i-1$.

Figure 13 illustrates how the frontier of labeled cells grows in this process, for $i = 0, 1, 2, 3$.

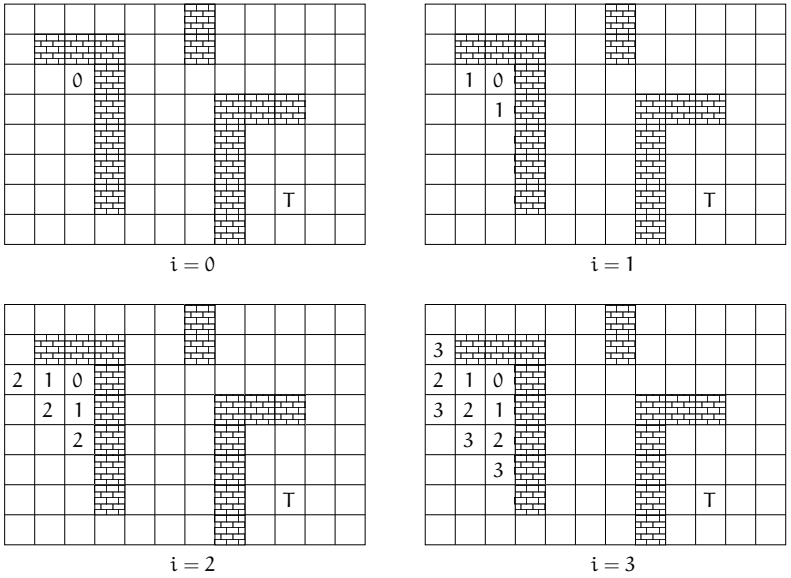


Figure 13: *The solution to subproblem* i *labels all cells* C *reachable from* S *within at most* i *steps with the length of the shortest path from* S *to* C.

Continuing in this fashion, we finally arrive at the situation depicted in Figure 14: all empty cells have been labeled (and are in fact reachable from S in this example). To find a shortest path from S to T, we start from T (which has label 21) and follow any path of decreasing labels $(20, 19, \ldots)$ until we finally reach S.

④	⑤	⑥	⑦	⑧	⑨		15	16	17	18	19
③				9	⑩		14	15	16	17	18
②	①	0		10	⑪	⑫	⑬	⑭	⑮	⑯	17
3	2	1		11	12	13				⑰	18
4	3	2		10	11	12		20	⑲	⑱	19
5	4	3		9	10	11		21	⑳	19	20
6	5	4		8	9	10		22	㉑	20	21
7	6	5	6	7	8	9		23	22	21	22

$i = 23$

Figure 14: *The solution to subproblem* $i = 23$ *solves the generalized problem and the original problem (a shortest path is obtained by starting from* T *and following a path of decreasing labels).*

The shortest path program. Let's get to the C++ implementation of the above method. We represent the floor by a dynamically allocated twodimensional array `floor` with dimensions $n+2$ and $m+2$ and entries of type `int`. (Formally, `floor` is a pointer to the first element of an array of $n+2$ pointers to `int`, but we still call this a twodimensional array). These dimensions leave space for extra walls surrounding the floor. Such extra walls allow us to get rid of special cases: floor cells having less than four adjacent cells. In general, an artificial data item that guards the actual data against special cases is called a *sentinel*.

The heart of the program (which appears as Program 16 below) is a loop that computes the solution to subproblem i from the solution to subproblem $i-1$, for $i = 1, 2, \ldots$. The solution to subproblem 0 is readily available: we set the `floor` entry corresponding to S to 0, and the entries corresponding to the empty cells to -1 (this is meant to indicate that the cell has not been labeled yet). Walls are always labeled with the integer -2.

In iteration i of the loop, we simply go through all the yet unlabeled cells and label exactly the ones with i that have an adjacent cell with label $i-1$. The loop terminates as soon as no progress is made anymore, meaning that no new cell could be labeled in the current iteration. Here is the code.

```
// main loop: find and label cells reachable in i=1,2,... steps
for (int i=1;; ++i) {
  bool progress = false;
  for (int r=1; r<n+1; ++r)
     for (int c=1; c<m+1; ++c) {
      if (floor[r][c] != -1) continue; // wall, or labeled before
      // is any neighbor reachable in i-1 steps?
      if (floor[r-1][c] == i-1 || floor[r+1][c] == i-1 ||
```

```
          floor[r][c-1] == i-1 || floor[r][c+1] == i-1 ) {
        floor[r][c] = i; // label cell with i
        progress = true;
      }
    }
  if (!progress) break;
}
```

The other parts of the `main` function are more or less straightforward. Initially, we read the dimensions from standard input and do the dynamic allocation.

```
// 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** floor = new int*[n+2];
for (int r=0; r<n+2; ++r)
  floor[r] = new int[m+2];
```

Next, we read the floor plan from standard input. We assume that it is given rowwise as a sequence of nm characters, where `'S'` and `'T'` stand for source and target, `'X'` represents a wall, and `'-'` an empty cell. The input file for our initial example from Figure 12 would then look as in Figure 15

```
8 12
------X-----
-XXX--X-----
--SX--------
---X---XXX--
---X---X----
---X---X----
---X---X-T--
-------X----
```

Figure 15: *Input for Program 16 corresponding to the example of Figure 12*

If other characters are found in the input (or if the input prematurely becomes empty), we generate empty cells. While reading the floor plan, we put the appropriate integers into the entries of `floor`, and we remember the target position for later.

```
// 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;
    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;
  }
```

Now we add the surrounding walls as sentinels.

```
// 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;
```

Next comes the main loop that we have already discussed above. It labels all reachable cells, so that we obtain a labeling as in Figure 14. From this labeling, we must now extract the shortest path from S to T. As explained above, this can be done by following a chain of adjacent cells with decreasing labels. For every cell on this path (except S), we put the integer -3 into the corresponding `floor` entry; this allows us to draw the path in the subsequent output. If no path was found (or if there is no target), the body of the `while` statement in the following code fragment is (correctly) not executed at all.

```
// 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) {
  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)
}
```

Finally, the output: we map the integer entries of `floor` back to characters, where -3 becomes `'o'`, our path symbol. Inserting `'\n'` at the right places, we obtain a copy of the input floor, with the shortest path appearing in addition. We must also not forget to delete the dynamically allocated arrays in the end.

```
// 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
for (int r=0; r<n+2; ++r)
  delete[] floor[r];
delete[] floor;

return 0;
```

In case of our initial example, the output looks like in Figure 16. Program 16 shows the complete source code.

```
oooooX-----
oXXX-oX-----
ooSX-oooooo-
---X---XXXo-
---X---X-oo-
---X---X-o--
---X---X-T--
-------X----
```

Figure 16: *Output of Program 16 on the input of Figure 15*

```
 1 #include<iostream>
 2
 3 int main()
 4 {
 5   // read floor dimensions
 6   int n; std::cin >> n; // number of rows
 7   int m; std::cin >> m; // number of columns
 8
 9   // dynamically allocate twodimensional array of dimensions
10   // (n+2) x (m+2) to hold the floor plus extra walls around
11   int** floor = new int*[n+2];
12   for (int r=0; r<n+2; ++r)
```



```
13     floor[r] = new int[m+2];
14
15   // target coordinates, set upon reading 'T'
16   int tr = 0;
17   int tc = 0;
18
19   // assign initial floor values from input:
20   // source:     'S'  ->     0 (source reached in 0 steps)
21   // target:     'T'  ->    -1 (number of steps still unknown)
22   // wall:       'X'  ->    -2
23   // empty cell: '-'  ->    -1 (number of steps still unknown)
24   for (int r=1; r<n+1; ++r)
25     for (int c=1; c<m+1; ++c) {
26       char entry = '-';
27       std::cin >> entry;
28       if      (entry == 'S') floor[r][c] = 0;
29       else if (entry == 'T') floor[tr = r][tc = c] = -1;
30       else if (entry == 'X') floor[r][c] = -2;
31       else if (entry == '-') floor[r][c] = -1;
32     }
33
34   // add surrounding walls
35   for (int r=0; r<n+2; ++r)
36     floor[r][0] = floor[r][m+1] = -2;
37   for (int c=0; c<m+2; ++c)
38     floor[0][c] = floor[n+1][c] = -2;
39
40   // main loop: find and label cells reachable in i=1,2,... steps
41   for (int i=1;; ++i) {
42     bool progress = false;
43     for (int r=1; r<n+1; ++r)
44       for (int c=1; c<m+1; ++c) {
45         if (floor[r][c] != -1) continue; // wall, or labeled before
46         // is any neighbor reachable in i-1 steps?
47         if (floor[r-1][c] == i-1 || floor[r+1][c] == i-1 ||
48             floor[r][c-1] == i-1 || floor[r][c+1] == i-1 ) {
49           floor[r][c] = i; // label cell with i
50           progress = true;
51         }
52       }
53     if (!progress) break;
54   }
55
56   // mark shortest path from source to target (if there is one)
57   int r = tr; int c = tc; // start from target
```

```
58    while (floor[r][c] > 0) {
59      int d = floor[r][c] - 1; // distance one less
60      floor[r][c] = -3; // mark cell as being on shortest path
61      // go to some neighbor with distance d
62      if      (floor[r-1][c] == d) --r;
63      else if (floor[r+1][c] == d) ++r;
64      else if (floor[r][c-1] == d) --c;
65      else                         ++c; // (floor[r][c+1] == d)
66    }
67
68    // print floor with shortest path
69    for (int r=1; r<n+1; ++r) {
70      for (int c=1; c<m+1; ++c)
71        if      (floor[r][c] == 0)   std::cout << 'S';
72        else if (r == tr && c == tc) std::cout << 'T';
73        else if (floor[r][c] == -3)  std::cout << 'o';
74        else if (floor[r][c] == -2)  std::cout << 'X';
75        else                         std::cout << '-';
76      std::cout << "\n";
77    }
78
79    // delete dynamically allocated arrays
80    for (int r=0; r<n+2; ++r)
81      delete[] floor[r];
82    delete[] floor;
83
84    return 0;
85  }
```

Program 16: *progs/shortest_path.C*

2.6.12 Beyond arrays and pointers

Arrays are very useful for many tasks and allow us to solve nontrivial problems like finding shortest paths in the previous section. From a theoretical point of view, arrays are in fact the only containers that we need.

On the other hand, there are two main drawbacks of arrays that we want to recapitulate here.

Arrays have fixed length. Any array, even if it is dynamically allocated, has a fixed length. In other words, we have to know *before* defining or dynamically allocating an array how many elements we need to store in it. Often, this is unrealistic. For example, in some application we might need to store a sequence of input numbers, but we don't know in advance how many numbers we will get. A typical "solution" is to dynamically allocate a very large array and just hope that the sequence fits in. The problems with this and a better (but still cumbersome) solution are outlined in Exercise 65.

A "real" solution is possible in C++ through the use of *vectors*. These are containers from the standard library that combine the classical array functionality (and its efficiency) with the possibility of growing (and shrinking) in length. Vectors can be implemented on top of arrays, and they have something similar to the mechanism outlined in Exercise 65 "built in". Vectors also largely remove the necessity of working with pointers. We will get to vectors (and their realization) later in this book.

Arrays are insecure. The usage of out-of-bound array indices is not detected in C++, and the same holds for pointers to addresses where no program object lives. With some care, you can write small programs that use arrays and pointers in a correct manner, but in complex programs, this is not easy at all. Debugging facilities of modern compilers can help, but even well-tested and frequently used large programs do not necessarily get it right. In fact, some people (let's call them attackers) are making a business of exploiting programming errors related to arrays and pointers in order to create malicious software.

Suppose that the attacker knows that some program—think of an operating system routine or a webserver—may (unintentionally) write input data beyond the bounds of an array. Due to the von-Neumann architecture, the part of the main memory being accidentally modified in this way may contain the actual program instructions. The attacker may then be able to prepare an input to the program in such a way that the program modifies itself to do whatever the attacker wants it to do. This modification runs with the same access rights as the original one, and these might be administrator rights in the worst case.

In this way, an attacker could "hijack" the computer that runs the program, and subsequently misuse it for illegal activities like sending spam, or paralyzing web servers by flooding them with requests.

For us that we are not (yet) professional programmers, the security aspect is less of a concern here. More important is that programming errors due to improper use of arrays and pointers can be very hard to find and often remain undetected until they suddenly result in strange and seemingly inexplicable behavior of the program. Also here, using vectors instead of arrays helps, since there are many potential errors related to arrays and pointers that you simply cannot make with vectors.

Why arrays, after all? Now you may ask why we have introduced arrays and pointers at all when there are more flexible and safer alternatives. Here are the three reasons.

1. Arrays and pointers are the simplest models of important standard library concepts (container and iterator).

2. Unlike vectors, arrays can be introduced without the need to discuss syntactical and semantical aspects of C++ functions and classes (that we simply don't have at our disposal at this point);

3. In order to really understand later how standard library containers and iterators are realized, it is necessary to know about arrays and pointers.

The take-home message here is this: it is important to get familiar with the *concepts* behind arrays and pointers, but it is less important to be able to actually program with arrays and pointers on a large scale.

2.6.13 Details

Command line arguments. In Program 15 for string matching, it is not very convenient that the search string is *fixed*. We then have to recompile the program every time we want to search for another string.

A more flexible alternative is to pass the search string as a *command line argument* that we provide upon calling the program.

The main function can access such command line arguments if we provide suitable parameters. Here is how the first ten lines of Program 15 have to be changed in order to make this work.

```
1  // Program: string_matching2.C
2  // find the first occurrence of a string (provided as command
3  // line argument) within the input text, and output text so far
4
5  #include<iostream>
6
7  int main (int argc, char* argv[])
8  {
9    if (argc < 2) {
10     // no command line arguments (except program name)
11     std::cout << "Usage: string_matching2 <string>\n";
12     return 1;
13   }
14
15   // search string: second command line argument
16   char* s = argv[1];
```

The values of `argc` and and `argv[]` (which is an array of pointers each of which in turn points to the first elements of a zero-terminated array of characters) are initialized by the operating system when it calls the `main` function. We will explain function parameters in detail later, here we will be satisfied with an example. Suppose that we call the program like this (assuming a Unix-type system):

```
./string_matching2 bool
```

Then `argc` (which counts the number of command line arguments) gets initialized with value 2. This count includes the program name itself (`"string_matching2"` in this case), and any additional strings provided on the command line (just the single string `"bool"` in this case). The 2 arrays `argv[0]` and `argv[1]` get initialized with the strings

`"string_matching2"` and `"bool"` as described in Section 2.6.10 above. Consequently, after its definition, the pointer variable `s` in the above piece of code points to the first element of a zero-terminated array of characters that corresponds to the string `"bool"`. This gets us back to the situation in Program 15 after line 10, and the remainders of both programs are identical.

2.6.14 Goals

Dispositional. At this point, you should ...

1) know what an array is, and what random access and iteration mean in the context of arrays;

2) understand the pointer concept, and how to compute with addresses;

3) be aware that (and understand why) arrays and pointers must be used with care;

4) know that characters and arrays of characters can be used to perform basic text processing tasks;

5) know that (multidimensional) arrays of variable length can be obtained by dynamic memory allocation;

Operational. In particular, you should be able to ...

(G1) read, understand, and argue about simple programs involving arrays and pointers;

(G2) write programs that define array variables or dynamically allocate (multidimensional) arrays;

(G3) write programs that read a sequence of data into a (dynamically allocated / multidimensional) array;

(G4) write programs that perform simple data processing tasks by using random access in (multidimensional) arrays as the major tool;

(G5) within programs, iterate over a (dynamically allocated / multidimensional) array by using pointer arithmetic;

(G6) write programs that perform simple text processing tasks with arrays of characters;

2.6.15 Exercises

Exercise 55

a) What does the following program output, and why?

```
#include<iostream>

int main()
{
  int a[] = {5, 6, 2, 3, 1, 4, 0};
  int* p = a;
  do {
    std::cout << *p << " ";
    p = a + *p;
  } while (p != a);

  return 0;
}
```

b) More generally, suppose that in the previous program, a *is initialized with some sequence of* n different *numbers in* $\{0,\dots,n-1\}$ *(we see this for* $n=7$ *in the previous program). Prove that the program terminates in this case.*

(G1)

Exercise 56 *Assume that in some program,* a *is an array of underlying type* int *and length* n.

a) Given a variable i *of type* int *with value* $0 \le i \le n$*, how can you obtain a pointer* p *to the element of index* i *in* a*? (Note: if* $i = n$*, this is asking for a past-the-end pointer.)*

b) Given a pointer p *to some element in* a*, how can you obtain the index* i *of this element? (Note: if* p *is a past-the-end pointer, the index is defined as* n*.)*

Write code fragments that compute p *from* i *in a) and* i *from* p *in b).* (G1)

Exercise 57 *Let us call a natural number* k-composite *if and only if it is divisible by exactly* k *different prime numbers. For example, prime powers are 1-composite, and* $6 = 2\cdot3$ *as well as* $20 = 2\cdot2\cdot5$ *are 2-composite. Write a program* k_composite.C *that reads numbers* $n \ge 0$ *and* $k \ge 0$ *from the input and then outputs all* k*-composite numbers in* $\{2,\dots,n-1\}$*. How many 7-composite numbers are there for* $n = 1,000,000$*?* (G2)(G4)

Exercise 58 *Write a program* invert.C *that inverts a* 3×3 *matrix* A *with real entries. The program should read the nine matrix entries from the input, and then output the inverse matrix* A^{-1} *(or the information that the matrix* A *is not invertible). In addition, the program should output the matrix* AA^{-1} *in order to let the user check whether the computation of the inverse was accurate (in the fully accurate case, the latter product is the identity matrix).*

Hint: *For the computation of the inverse, you can employ* Cramer's rule*. Applied to the computation of the inverse, it yields that* A^{-1}_{ij} *(the entry of* A^{-1} *in row* i *and columns* j*) is given by*

$$A^{-1}_{ij} = \frac{(-1)^{i+j}\det(A^{ji})}{\det(A)},$$

where $\det(M)$ *is the determinant of a square matrix* M*, and* A^{ij} *is the* 2×2 *matrix obtained from* A *by deleting row* j *and column* i*.*

To compute the determinant of a 3×3 *matrix, you might want to use the well-known* Sarrus' rule*.* (G2)(G3)(G4)

Exercise 59 *Write a program* read_array *that reads a sequence of* n *integers from standard input into an array. The number* n *is the first input, and then the program expects you to input another* n *values. After reading the* n *values, the program should output them in the same order. (If you can do this, you have proven that you are no longer a complete novice, according to Stroustrup.) For example, on input* 5 4 3 6 1 2 *the program should output* 4 3 6 1 2*.* (G2)(G3)

Exercise 60 *Enhance the program* read_array.C *from Exercise 59 so that the resulting program* sort_array.C *sorts the array elements into ascending order before outputting them. Your sorting algorithm does not have to be particularly efficient, the main thing here is that it works correctly. Test your program on some larger inputs (preferably read from a file, after redirecting standard input). For example, on input* 5 4 3 6 1 2 *the program should output* 1 2 3 4 6*.* (G2)(G3)(G4)

Exercise 61 *Enhance the program* read_array.C *from Exercise 59 so that the resulting program* cycles.C *interprets the input sequence of* n *integers as a* permutation π *of* $\{0,\dots,n-1\}$*, and that it outputs the* cycle decomposition *of* π*.*

Some explanations are in order: a permutation π *is a bijective mapping from the set* $\{0,\dots,n-1\}$ *to itself; therefore, the input sequence can be interpreted as the sequence of values* $\pi(0),\dots,\pi(n-1)$ *of a permutation* π *if and only if it contains every number from* $\{0,\dots,n-1\}$ *exactly once.*

The program cycles.C *should first check whether the input sequence satisfies this condition, and if not, terminate with a corresponding message. If the input indeed encodes a permutation* π*, the program should output the* cycle decomposition *of* π*. A cycle in* π *is any sequence of the form* $(\ n_1\ n_2\ \cdots\ n_k\ )$ *such that*

- $n_2 = \pi(n_1),\quad n_3 = \pi(n_2),\quad \dots,\quad n_k = \pi(n_{k-1})$*, and* $n_1 = \pi(n_k)$*, and*
- n_1 *is the smallest element among* $n_1,\dots,n_k$*.*

Any cycle uniquely determines the π*-values of all its elements; on the other hand, every element appears in some cycle (which might be of the trivial form* (n_1)*, meaning that* $\pi(n_1) = n_1$*). This implies that the permutation decomposes into a unique*

set of cycles. For example, the permutation π *given by*

$$\pi(0)=4,\quad \pi(1)=2,\quad \pi(2)=3,\quad \pi(3)=1,\quad \pi(4)=0$$

decomposes into the two cycles (0 4) *and* (1 2 3). (G2)(G3)(G4)

Exercise 62 *Consider the string matching algorithm of Program 15. Prove that for all* $m > 1, n \geq m$*, there exists a search string* s *of length* m *and a text* t *of length* n *on which the algorithm in Program 15 performs* $m(n-m+1)$ *comparisons between single characters.* (G1)

Exercise 63 *Consider the following program that defines and initializes a threedimensional 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]
      }
    };

  return 0;

}
```

Write a program `threedim_array.C` *that enhances this program by a (nested) loop that iterates over the array* a *and its subarrays to output all the 24* `int` *values that are stored in* a *and its subarrays. Do not use random access to do this but pointer arithmetic.* (G5)

Exercise 64 *Write a program* `frequencies.C` *that reads a text from standard input (like in Program 15) and outputs the frequencies of the letters in the text, where we do not distinguish between lower and upper case letters. For this exercise, you may assume that the type* `char` *implements ASCII encoding. This means that all characters have integer values in* $\{0,1,\ldots,127\}$*. Moreover, in ASCII, the values of the 26 upper case literals* `'A'` *up to* `'Z'` *are consecutive numbers in* $\{65,\ldots,90\}$*; for the lower case literals* `'a'` *up to* `'z'`*, the value range is* $\{97,\ldots,122\}$. (G6)

Running this on the lyrics of *Yesterday* (The Beatles) for example should yield the following output.

```
Frequencies:
a:     45 of 520
b:     5 of 520
c:     5 of 520
d:     28 of 520
e:     65 of 520
f:     4 of 520
g:     13 of 520
h:     27 of 520
i:     27 of 520
j:     0 of 520
k:     3 of 520
l:     20 of 520
m:     10 of 520
n:     30 of 520
o:     43 of 520
p:     4 of 520
q:     0 of 520
r:     19 of 520
s:     36 of 520
t:     31 of 520
u:     9 of 520
v:     6 of 520
w:     19 of 520
x:     0 of 520
y:     34 of 520
z:     0 of 520
Other: 37 of 520
```

2.6.16 Challenges

Exercise 65 *The fact that an array has fixed length is often inconvenient. For example, in Exercise 59 and in Exercise 60, the number of elements to be read into the array had to be provided as the first input in order for the program to be able to dynamically allocate an array of the appropriate length. But in practice, the length of the input sequence is often not known a priori.*

We would therefore like to write a program that reads a sequence of integers from standard input into an array, where the length of the sequence is not known beforehand (and not part of the input)—the program should simply read one number after another until the stream becomes empty.

One possible strategy is to dynamically allocate an array of large length, big enough to store any possible input sequence. But if the sequence is short, this is a huge waste of memory, and if the sequence is very long, the array might still not be large enough.

a) Write a program `read_array2.C` *that reads a sequence of integers of unknown length into an array, and then outputs the sequence. The program should satisfy the following two properties.*

(i) The amount of dynamically allocated memory in use by the program should at any time be proportional to the number of sequence elements that have been read so far. *To be concrete: there must be a positive constant* a

such that no more than ak *cells of dynamically allocated memory are in use when* k *elements have been read,* $k \geq 1$*. We refer to this property as* space efficiency*. It ensures that even very long sequences can be read (up to the applicable memory limits), but that short sequences consume only little memory.*

(ii) The number of assignments (of values to array elements) performed so far should at any time be proportional to the number of sequence elements that have been read so far, with the same meaning of proportionality as above. We refer to this property as time efficiency*. It ensures that the program is only by a constant factor slower than the program* `read_array.C` *that knows the sequence length in advance.*

b) Determine the constants of proportionality a *for properties (i) and (ii) of your program.*

Exercise 66 *For larger floors, Program 16 can become quite inefficient, since every step* i *examines* all *cells of the floor in order to find the (possibly very few) ones that have to be labeled with* i *in that step. A better solution would be to examine only the neighbors of the cells that are already labeled with* $i-1$*, since only these are candidates for getting label* i*.*

Write a program `shortest_path_fast.C` *that realizes this idea, and measure the performance gain on some larger floors of your choice.*

Exercise 67 *The* `XBM` *file format is a format for storing monochrome (black & white) images. The format is somewhat outdated, but many browsers (Internet Explorer is a notable exception) can still display images in* `XBM` *format.*

An `XBM` *image file for an image named* `test` *might look like this (taken from Wikipedia's* `XBM` *page).*

```
#define test_width 16
#define test_height 7
static char test_bits[] = {
  0x13, 0x00, 0x15, 0x00, 0x93, 0xcd, 0x55,
  0xa5, 0x93, 0xc5, 0x00, 0x80, 0x00, 0x60};
```

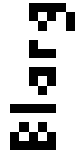
As you can guess from this, `XBM` *files are designed to be integrated into C and C++ source code which makes it easy to process them (there is no need to read in the data; simply include the file from the C++ program that needs to process the image). In our example,* `test_width` *and* `test_height` *denote the width and height of the image in pixels. Formally, these names are* macros*, but in the program they can be used like constant expressions.* `test_bits` *is an array of characters that encodes the colors of the* 16×7 *pixels in the image. Every* hexadecimal literal *of the form* $0xd_1d_2$ *encodes eight pixels, where the order is row by row. In our case,* `0x13` *and* `0x00` *encode the 16 pixels of the first row, while* `0x15` *and* `0x00` *are for the second row, etc.*

Here is how a two-digit hexadecimal literal encodes the colors of eight consecutive pixels within a row.[26] *Every hexadecimal digit* d_i *is from the set* $\{0,...,9,a,\ldots,f\}$ *where* a *up to* f *stand for* $10,\ldots,15$*. The actual number encoded by a hexadecimal literal is* $16d_1 + d_2 \in \{0,\ldots,255\}$.[27] *For example,* `0x13` *has value* $1 \cdot 15 + 3 = 19$*.*

Now, any number in $\{0,\ldots,255\}$ *has a binary representation with 8 bits. 19, for example, has binary representation* 00010011*. The pixel colors are obtained by reading this backwards, and interpreting* 1 *as black and* 0 *as white. Thus, the first eight pixels in row* 1 *of the* `test` *image are black, black, white, white, black, white, white, white. The complete* `test` *image looks like this:*

Write a program `xbm.C` *that* `#include`*s an* `XBM` *file of your choice (you may search the web to find suitable* `XBM` *files), and that outputs an* `XBM` *file for the same image,* rotated by 90 degrees*. The program may write the resulting file to standard output. In case of the* `test` *image, the resulting* `XBM` *file and the resulting rotated image are as follows.*

```
#define rotated_width 7
#define rotated_height 16
static char rotated_bits[] = {
   0x3c, 0x54, 0x48, 0x00,
   0x04, 0x1c, 0x00, 0x1c,
   0x14, 0x08, 0x00, 0x1f,
   0x00, 0x0a, 0x15, 0x1f};
```



Note that we now have 16 instead of 14 hexadecimal literals. This is due to the fact that each of the 16 rows needs one literal for its 7 pixels, where the leading bits of the binary representations are being ignored.

You may extend your program to perform other kinds of image processing tasks of your choice. Examples include color inversion (replace black with white, and vice versa), computing a mirror image, scaling the image (so that it occupies less or more pixels), etc.

[26] If the width is not a multiple of 8, the superfluous color values from the last hexadecimal literal of each row are being ignored.

[27] If the type `char` has value range $\{-128,\ldots,127\}$, the silent assumption is that a literal value α larger than 127 converts to $\alpha - 256$, which has the same representation under two's complement.