
7.2. Arrays

An array — an instance of the container class `array<>` — models a static array. It wraps an ordinary static C-style array providing the interface of an STL container (Figure 7.1). Conceptually, an array is a sequence of elements with constant size. Thus, you can neither add nor remove elements to change the size. Only a replacement of element values is possible.

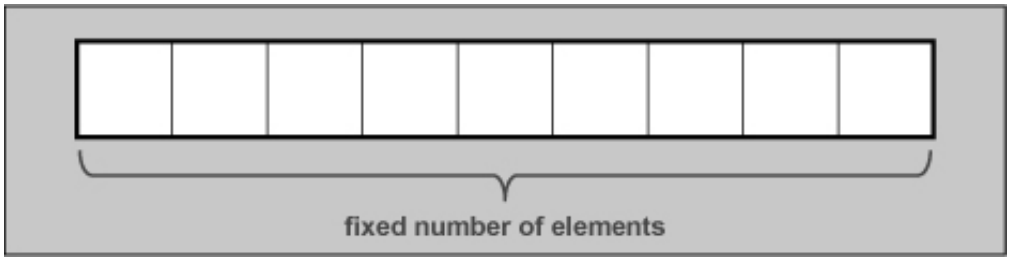

Figure 7.1. Structure of an Array

Class `array<>`, introduced to the C++ standard library with TR1, results from a useful wrapper class for ordinary C-style arrays Bjarne Stroustrup introduced in his book [*Stroustrup:C++*]. It is safer and has no worse performance than an ordinary array.

To use an array, you must include the header file `<array>`:

```
#include <array>
```

There, the type is defined as a class template inside namespace `std`:

```
namespace std {
    template <typename T, size_t N>
    class  array;
}
```

The elements of an array may have any type `T`.

The second template parameter specifies the number of elements the array has throughout its lifetime. Thus, `size()` always yields `N`.

Allocator support is not provided.

7.2.1. Abilities of Arrays

Arrays copy their elements into their internal static C-style array. The elements always have a certain order. Thus, arrays are a kind of *ordered collection*. Arrays provide *random access*. Thus, you can access every element directly in constant time, provided that you know its position. The iterators are random-access iterators, so you can use any algorithm of the STL.

If you need a sequence with a fixed number of elements, class `array<>` has the best performance because memory is allocated on the stack (if possible), reallocation never happens, and you have random access.

Initialization

Regarding initialization, class `array<>` has some unique semantics. As a first example, the default constructor does not create an empty container, because the number of elements in the container is always constant according to the second template parameter throughout its lifetime.

Note that `array<>` is the only container whose elements are *default initialized* when nothing is passed to initialize the elements. This means that for fundamental types, the initial value might be undefined rather than zero (see Section 3.2.1, page 37). For example:

```
std::array<int,4> x;          // OOPS: elements of x have undefined value
```

You can provide an empty initializer list instead. In that case, all values are guaranteed to be value initialized, which has the effect that elements of fundamental types are *zero initialized*:

```
std::array<int,4> x = {};     // OK: all elements of x have value 0 (int())
```

The reason is that although `array<>` seems to provide a constructor for initializer lists, it does not. Instead, `array<>` fulfills the requirements of an aggregate.[3] Therefore, even before C++11, you could use an initializer list to initialize an array when it got created:

[3] An aggregate is an array or a class with no user-provided constructors, no private or protected

nonstatic data members, no base classes, and no virtual functions.

```
std::array<int,5> coll = { 42, 377, 611, 21, 44 };
```

The elements in the initializer list must have the same type, or there must be a type conversion to the element type of the array defined.

If an initializer list does not have enough elements, the elements in the array are initialized via the default constructor of the element type. In this case, it is guaranteed that for fundamental data types the elements are zero initialized. For example:

Click here to view code image

```
std::array<int,10> c2 = { 42 };  // one element with value 42
                                 // followed by 9 elements with value 0
```

If the number of elements in the initializer lists is higher than the size of the array, the expression is ill-formed:

```
std::array<int,5> c3 = { 1, 2, 3, 4, 5, 6 }; // ERROR: too many values
```

Because no constructors or assignment operators for initializer lists are provided, initializing an array during its declaration is the only way to use initializer lists. For this reason, you also can't use the parenthesis syntax to specify initial values (which differs from other container types):

```
std::array<int,5> a({ 1, 2, 3, 4, 5, 6 });     // ERROR
std::vector<int>  v({ 1, 2, 3, 4, 5, 6 });     // OK
```

Class `array<>` being an aggregate also means that the member that holds all the elements is public. However, its name is not specified in the standard; thus, any direct access to the public member that holds all elements results in undefined behavior and is definitely not portable.

`swap()` and Move Semantics

As for all other containers, `array<>` provides `swap()` operations. Thus, you can swap elements with a container of the same type (same element type and same number of elements). Note, however, that an `array<>` can't simply swap pointers internally. For this reason, `swap()` has linear complexity and the effect that iterators and references don't swap containers with their elements. So, iterators and references refer to the same container but different elements afterward.

You can use move semantics, which are implicitly provided for arrays. For example:[4]

[4] Thanks to Daniel Krügler for providing this example.

```
std::array<std::string,10>   as1, as2;
...
as1 = std::move(as2);
```

Size

It is possible to specify a size of `0`, which is an array with no elements. In that case, `begin()` and `end()`, `cbegin()` and `cend()`, and the corresponding reverse iterators still yield the same unique value. However, the return value of `front()` and `back()` is undefined:

Click here to view code image

```
std::array<Elem,0> coll;                  // array with no elements

std::sort(coll.begin(),coll.end());       // OK (but has no effect)

coll[5] = elem;                           // RUNTIME ERROR  undefined
behavior
std::cout << coll.front();                // RUNTIME ERROR  undefined
behavior
```

For `data()`, the return value is unspecified, which means that you can pass the return value to other places as long as you don't dereference it.

7.2.2. Array Operations

Create, Copy, and Destroy

Table 7.4 lists the constructors and destructors for arrays. Because class `array<>` is an aggregate, these operations are only implicitly defined. You can create arrays with and without elements for initialization. The default constructor default initializes the elements, which means that the value of fundamental types is undefined. If you use an initializer list but do not pass enough elements, the remaining elements are created with their default constructor (`0` for fundamental types). See Section 7.1.2, page 254, for some remarks about possible initialization sources.

Table 7.4. Constructors and Destructor of Class `array<>`

Operation	Effect
`array<Elem,N> c`	Default constructor; creates an array with default-initialized elements
`array<Elem,N> c(c2)`	Copy constructor; creates a copy of another array of the same type (all elements are copied)
`array<Elem,N> c = c2`	Copy constructor; creates a copy of another array of the same type (all elements are copied)
`array<Elem,N> c(rv)`	Move constructor; creates a new array taking the contents of the rvalue *rv* (since C++11)
`array<Elem,N> c = rv`	Move constructor; creates a new array, taking the contents of the rvalue *rv* (since C++11)
`array<Elem,N> c = initlist`	Creates an array initialized with the elements of the initializer list

Again, note that unlike with other containers, you can't use the parenthesis syntax with initializer lists:

```
std::array<int,5> a({ 1, 2, 3, 4, 5 }); // ERROR
```

Nonmodifying Operations

Table 7.5 lists all nonmodifying operations of arrays. See the additional remarks in Section 7.1.2, page 254.

Table 7.5. Nonmodifying Operations of Class `array<>`

Operation	Effect
`c.empty()`	Returns whether the container is empty (equivalent to `size()==0` but might be faster)
`c.size()`	Returns the current number of elements
`c.max_size()`	Returns the maximum number of elements possible
`c1 == c2`	Returns whether *c1* is equal to *c2* (calls == for the elements)
`c1 != c2`	Returns whether *c1* is not equal to *c2* (equivalent to `!(c1==c2)`)
`c1 < c2`	Returns whether *c1* is less than *c2*
`c1 > c2`	Returns whether *c1* is greater than *c2* (equivalent to `c2<c1`)
`c1 <= c2`	Returns whether *c1* is less than or equal to *c2* (equivalent to `!(c2<c1)`)
`c1 >= c2`	Returns whether *c1* is greater than or equal to *c2* (equivalent to `!(c1<c2)`)

Assignments

Table 7.6 lists the ways to assign new values. Besides the assignment operator, you can use only `fill()` to assign a new value to each element, or `swap()` to swap values with another array. For operator `=` and `swap()`, both arrays have to have the same type, which means that both element type and size have to be the same.

Table 7.6. Assignment Operations of Class `array<>`

Operation	Effect
`c = c2`	Assigns all elements of *c2* to *c*
`c = rv`	Move assigns all elements of the rvalue *rv* to *c* (since C++11)
`c.fill(val)`	Assigns *val* to each element in array *c*
`c1.swap(c2)`	Swaps the data of *c1* and *c2*
`swap(c1,c2)`	Swaps the data of *c1* and *c2*

Note that `swap()` can't guarantee constant complexity for arrays, because it is not possible to exchange some pointers internally (see Section 7.2.1, page 263). Instead, as with the algorithm `swap_ranges()` (see Section 11.6.4, page 566), for both arrays involved, all elements get new values assigned.

Internally, all these operations call the assignment operator of the element type.

Element Access

To access all elements of an array, you must use range-based `for` loops (see Section 3.1.4, page 17), specific operations, or iterators. In addition, a tuple interface is provided, so you can also use `get<>()` to access a specific element (see Section 7.2.5, page 268, for

details). Table 7.7 shows all array operations for direct element access. As usual in C and C++, the first element has index `0`, and the last element has index `size()-1`. Thus, the *n*th element has index *n* `-1`. For nonconstant arrays, these operations return a reference to the element. Thus, you could modify an element by using one of these operations, provided it is not forbidden for other reasons.

Table 7.7. Direct Element Access of Class `array<>`

Operation	Effect
`c[idx]`	Returns the element with index *idx* (*no* range checking)
`c.at(idx)`	Returns the element with index *idx* (throws range-error exception if *idx* is out of range)
`c.front()`	Returns the first element (*no* check whether a first element exists)
`c.back()`	Returns the last element (*no* check whether a last element exists)

The most important issue for the caller is whether these operations perform range checking. Only `at()` performs range checking. If the index is out of range, `at()` throws an `out_of_range` exception (see Section 4.3, page 41). All other functions do *not* check. A range error results in undefined behavior. Calling operator `[]`, `front()`, and `back()` for an empty `array<>` always results in undefined behavior. Note however that it is only empty if declared to have a size of 0:

Click here to view code image

```
std::array<Elem,4> coll;        // only four elements!

coll[5] = elem;                 // RUNTIME ERROR      undefined behavior
std::cout << coll.front();      // OK (coll has 4 element after construction)
std::array<Elem,0> coll2;       // always empty
std::cout << coll2.front();     // RUNTIME ERROR      undefined behavior
```

So, if in doubt, you must ensure that the index for operator `[]` is valid or use `at()`:

```
template <typename C>
void foo (C& coll)
{
    if (coll.size() > 5) {
        coll[5] = ...;          // OK
    }

    coll.at(5) = ...;           // throws out_of_range exception for
array<...,4>
}
```

Note that this code is OK only in single-threaded environments. In multithreaded contexts, you need synchronization mechanisms to prevent `coll` from being modified between the check for its size and the access to the element (see Section 18.4.3, page 984, for details).

Iterator Functions

Arrays provide the usual operations to get iterators (Table 7.8). Array iterators are random-access iterators (see Section 9.2, page 433, for a discussion of iterator categories). Thus, in principle, you could use all algorithms of the STL.

Table 7.8. Iterator Operations of Class `array<>`

Operation	Effect
`c.begin()`	Returns a random-access iterator for the first element
`c.end()`	Returns a random-access iterator for the position after the last element
`c.cbegin()`	Returns a constant random-access iterator for the first element (since C++11)
`c.cend()`	Returns a constant random-access iterator for the position after the last element (since C++11)
`c.rbegin()`	Returns a reverse iterator for the first element of a reverse iteration
`c.rend()`	Returns a reverse iterator for the position after the last element of a reverse iteration
`c.crbegin()`	Returns a constant reverse iterator for the first element of a reverse iteration (since C++11)
`c.crend()`	Returns a constant reverse iterator for the position after the last element of a reverse iteration (since C++11)

The exact type of these iterators is implementation defined. For arrays, however, the iterators returned by `begin()`,

`cbegin()`, `end()`, and `cend()` are often ordinary pointers, which is fine because an `array<>` internally uses a C-style array for the elements and ordinary pointers provide the interface of random-access iterators. However, you can't count on the fact that the iterators are ordinary pointers. For example, if a safe version of the STL that checks range errors and other potential problems is used, the iterator type is usually an auxiliary class. See Section 9.2.6, page 440, for a nasty difference between iterators implemented as pointers and iterators implemented as classes.

Iterators remain valid as long as the array remains valid. However, unlike for all other containers, `swap()` assigns new values to the elements that iterators, references, and pointers refer to.

7.2.3. Using `array`s as C-Style Arrays

As for class `vector<>`, the C++ standard library guarantees that the elements of an `array<>` are in contiguous memory. Thus, you can expect that for any valid index `i` in array `a`, the following yields `true`:

```
&a[i] == &a[0] + i
```

This guarantee has some important consequences. It simply means that you can use an `array<>` wherever you can use an ordinary C-style array. For example, you can use an array to hold data of ordinary C-strings of type `char*` or `const char*`:

Click here to view code image

```
std::array<char,41> a;              // create static array of 41 chars

strcpy(&a[0],"hello, world");       // copy a C-string into the array
printf("%s\n", &a[0]);              // print contents of the array as C-string
```

Note, however, that you don't have to use the expression `&a[0]` to get direct access to the elements in the array, because the member function `data()` is provided for this purpose:

Click here to view code image

```
std::array<char,41> a;              // create static array of 41 chars

strcpy(a.data(),"hello, world");    // copy a C-string into the array
printf("%s\n", a.data());           // print contents of the array as C-string
```

Of course, you have to be careful when you use an `array<>` in this way (just as you always have to be careful when using ordinary C-style arrays and pointers). For example, you have to ensure that the size of the array is big enough to copy some data into it and that you have an `'\0'` element at the end if you use the contents as a C-string. However, this example shows that whenever you need an array of type `T` for any reason, such as for an existing C library, you can use an `array<>` (or `vector<>`) and use `data()` where the ordinary C-style interface is required.

Note that you must not pass an iterator as the address of the first element. Iterators of class `array<>` have an implementation-specific type, which may be totally different from an ordinary pointer:

Click here to view code image

```
printf("%s\n", a.begin());          // ERROR (might work, but not portable)
printf("%s\n", a.data());           // OK
```

7.2.4. Exception Handling

Arrays provide only minimal support for logical error checking. The only member function for which the standard requires that it may throw an exception is `at()`, which is the safe version of the subscript operator (see Section 7.2.2, page 265).

For functions called by an array (functions for the element type or functions that are user-supplied) no special guarantees are generally given (because you can't insert or delete elements, exceptions might occur only if you copy, move, or assign values). Note especially that `swap()` might throw because it performs an element-wise swap, which might throw.

See Section 6.12.2, page 248, for a general discussion of exception handling in the STL.

7.2.5. Tuple Interface

Arrays provide the tuple interface (see Section 5.1.2, page 68). Thus, you can use the expressions `tuple_size<>::value` to yield the number of elements, `tuple_element<>::type` to yield the type of a specific element, and `get()` to gain access to a specific element. For example:

Click here to view code image

```
typedef std::array<std::string,5> FiveStrings;
```

```
FiveStrings a = { "hello", "nico", "how", "are", "you" };

std::tuple_size<FiveStrings>::value       // yields 5
std::tuple_element<1,FiveStrings>::type   // yields std::string
std::get<1>(a)                            // yields std::string("nico")
```

7.2.6. Examples of Using Arrays

The following example shows a simple use of class `array<>` :

```
// cont/array1.cpp

#include <array>
#include <algorithm>
#include <functional>
#include <numeric>
#include "print.hpp"
using namespace std;

int main()
{
    // create array with 10 ints
    array<int,10> a = { 11, 22, 33, 44 };

    PRINT_ELEMENTS(a);

    // modify last two elements
    a.back() = 9999999;
    a[a.size()-2] = 42;
    PRINT_ELEMENTS(a);

    // process sum of all elements
    cout << "sum: "
         << accumulate(a.begin(),a.end(),0)
         << endl;

    // negate all elements
    transform(a.begin(),a.end(),   // source
              a.begin(),           // destination
              negate<int>());      // operation
    PRINT_ELEMENTS(a);
}
```

As you can see, you can use the general container interface operations (operator `=` , `size()` , and operator `[]`) to manipulate the container directly. Because member functions such as `begin()` and `end()` for iterator access are also provided, you can also use different operations that call `begin()` and `end()` , such as modifying and nonmodifying algorithms and the auxiliary function `PRINT_ELEMENTS()` , which is introduced in Section 6.6, page 216.

The output of the program is as follows:

```
11 22 33 44 0 0 0 0 0 0
11 22 33 44 0 0 0 0 42 9999999
sum: 10000151
-11 -22 -33 -44 0 0 0 0 -42 -9999999
```