**Arrays**

An ordinary static C-style array providing the interface of an STL container

header file #include <array>

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

namespace std {

template <typename T, size\_t N>

class array;

}

Allocator support is not provided.

# Abilities of Arrays

The elements always have a certain order. Thus, arrays are a kind of ordered collection.

Arrays provide random access.

The iterators are random-access iterators, so you can use any algorithm of the STL.

## Container properties

### Sequence

Elements in sequence containers are ordered in a strict linear sequence.

### Contiguous storage

The elements are stored in contiguous memory locations, allowing constant time random access to elements. Pointers to an element can be offset to access other elements.

### Fixed-size aggregate

The container uses implicit constructors and destructors to allocate the required space statically. Its size is compile-time constant. No memory or time overhead.

## Initialization

Regarding initialization, class array<> has some unique semantics.

1. The default constructor does not create an empty container, because the number of elements in the container is always constant.
2. 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.

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

You can provide an empty initializer list instead

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.

1. Instead, array<> fulfills the requirements of an aggregate. Therefore, even before C++11, a programmer could use an initializer list to initialize an array when it got created:

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

An aggregate is an array or a class with no user-provided constructors, no private or protected non-static data members, no base classes, and no virtual functions.

1. If an initializer list does not have enough elements, the elements in the array are initialized via the default constructor of the element type.

std::array<int,10> c2 = { 42 }; // one element with value 42

// followed by 9 elements with value 0

1. 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

1. 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

You can swap elements with a container of the same type (same element type and same number of elements).

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.

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 case of a zero-length array (N == 0)

array.begin() == array.end(), which is some unique value.

In that case, begin() and end(), cbegin() and cend(), and the corresponding reverse iterators still yield the same unique value.

The effect of calling front() or back() on a zero-sized array is undefined.

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.

#include <iostream>

#include <array>

using namespace std;

int main(void) {

std::array<int, 0> a;

//cout << a.front() << endl; // Runtime Errors: Segmentation Fault (SIGSEGV)

//cout << a.back() << endl; // Runtime Errors: Segmentation Fault (SIGSEGV)

cout << a.data() << endl; // output: 0

return 0;

}

#include <iostream>

#include <array>

using namespace std;

int main(void) {

array<int, 5> a1;

array<int, 5> a2 = {1, 2, 3, 4, 5};

array<int, 5> a3(a2);

array<int, 5> a4 = a3;

//array<int, 5> a5 = {1, 2, 3, 4, 5, 6};

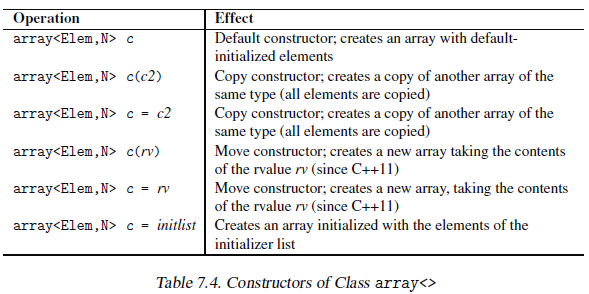
// CE error: too many initializers for 'std::array<int, 5ul>'

return 0;

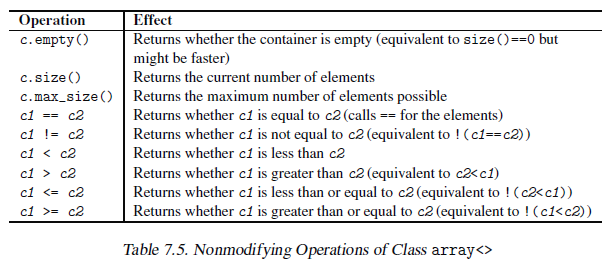
}

# Array Operations

## Create, Copy, and Destroy



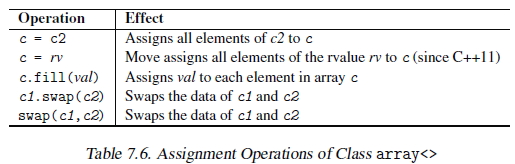
## Nonmodifying Operations



size and max\_size both return same value which is equal to the second template parameter used to instantiate the array template class (N).

no of elements i.e. std::distance(begin(), end())

## Assignments



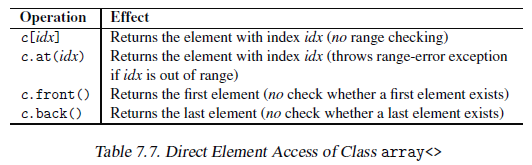
For operator = and swap(), both arrays have to have the same type, which means that both element type and size have to be the same.

Note that swap() can’t guarantee constant complexity for arrays, because it is not possible to exchange some pointers internally.

Instead, as with the algorithm swap\_ranges(), for both arrays involved, all elements get new values assigned.

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

## Element Access



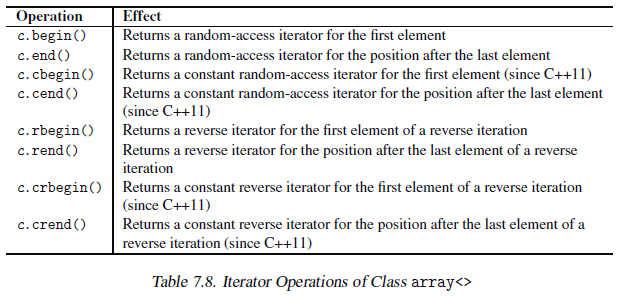
For nonconstant arrays, these operations return a reference to the element.

Thus, you could modify an element by using one of these operations, provided it is not forbidden for other reasons.

Only at() performs range checking. If the index is out of range, at() throws an out\_of\_range exception. 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.

## Iterator Functions



Array iterators are random-access iterators. Thus, in principle, you could use all algorithms of the STL.

**Iterator invalidation**

As a rule, iterators to an array are never invalidated throughout the lifetime of the array.

However, unlike for all other containers, swap() assigns new values to the elements that iterators, references, and pointers refer to.

The exact type of these iterators is implementation defined.

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.

# Using arrays as C-Style Arrays

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

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\*:

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~~

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

Do not 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.

**Note:**

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:

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

printf("%s\n", a.data()); // OK

# Exception Handling

Arrays provide only minimal support for logical error checking.

Only member function at() may throw an exception.

For functions called by an array 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.

# Tuple Interface

Arrays provide the tuple interface. An array can also be used as a tuple of N elements of the same type.

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")

#include <iostream>

#include <array>

using namespace std;

int main(void) {

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

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

cout << "size: " << std::tuple\_size<FiveStrings>::value << endl;

// yields 5

//cout << "type: " << std::tuple\_element<1,FiveStrings>::type << endl; // yields std::string

cout << "elem: " << std::get<1>(a) << endl;

//yields std::string("nico")

return 0;

}

# END