std::array

Defined in header <array>

```
template<
    class T,
    std::size_t N
> struct array;
```
(since C++11)

`std::array` is a container that encapsulates fixed size arrays.

This container is an aggregate type with the same semantics as a struct holding a C-style array `T[N]` as its only non-static data member. It can be initialized with aggregate-initialization, given at most `N` initializers that are convertible to `T`: `std::array<int, 3> a = {1,2,3};`

The struct combines the performance and accessibility of a C-style array with the benefits of a standard container, such as knowing its own size, supporting assignment, random access iterators, etc.

`std::array` satisfies the requirements of `Container` and `ReversibleContainer` except that default-constructed array is not empty and that the complexity of swapping is linear, `, ContiguousContainer)` (since C++17) and partially satisfies the requirements of `SequenceContainer`

There is a special case for a zero-length array (`N == 0`). In that case, `array.begin() == array.end()`, which is some unique value. The effect of calling `front()` or `back()` on a zero-sized array is undefined.

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

Iterator invalidation

As a rule, iterators to an array are never invalidated throughout the lifetime of the array. One should take note, however, that during swap, the iterator will continue to point to the same array element, and will thus change its value.

Member types

Member type	Definition
`value_type`	`T`
`size_type`	`std::size_t`
`difference_type`	`std::ptrdiff_t`
`reference`	`value_type&`
`const_reference`	`const value_type&`
`pointer`	`value_type*`
`const_pointer`	`const value_type*`
`iterator`	`RandomAccessIterator`
`const_iterator`	Constant random access iterator
`reverse_iterator`	`std::reverse_iterator<iterator>`
`const_reverse_iterator`	`std::reverse_iterator<const_iterator>`

Member functions

Implicitly-defined member functions

(constructor) (implicitly declared)	initialized the array following the rules of aggregate initialization (note that default initialization may result in indeterminate values for non-class `T`) (public member function)
(destructor) (implicitly declared)	destroys every element of the array

	(public member function)
operator= (implicitly declared)	overwrites every element of the array with the corresponding element of another array (public member function)

Element access

at	access specified element with bounds checking (public member function)
operator[]	access specified element (public member function)
front	access the first element (public member function)
back	access the last element (public member function)
data	direct access to the underlying array (public member function)

Iterators

begin **cbegin**	returns an iterator to the beginning (public member function)
end **cend**	returns an iterator to the end (public member function)
rbegin **crbegin**	returns a reverse iterator to the beginning (public member function)
rend **crend**	returns a reverse iterator to the end (public member function)

Capacity

empty	checks whether the container is empty (public member function)
size	returns the number of elements (public member function)
max_size	returns the maximum possible number of elements (public member function)

Operations

fill	fill the container with specified value (public member function)
swap	swaps the contents (public member function)

Non-member functions

operator== **operator!=** **operator<** **operator<=** **operator>** **operator>=**	lexicographically compares the values in the array (function template)
std::get(std::array)	accesses an element of an `array` (function template)
std::swap(std::array) (C++11)	specializes the `std::swap` algorithm (function template)

Helper classes

std::tuple_size<std::array>	obtains the size of an `array`

	(class template specialization)
std::tuple_element<std::array>	obtains the type of the elements of `array` (class template specialization)

Example

Run this code

```
#include <string>
#include <iterator>
#include <iostream>
#include <algorithm>
#include <array>

int main()
{
    // construction uses aggregate initialization
    std::array<int, 3> a1{ {1, 2, 3} }; // double-braces required in C++11 (not in C++14)
    std::array<int, 3> a2 = {1, 2, 3};  // never required after =
    std::array<std::string, 2> a3 = { std::string("a"), "b" };

    // container operations are supported
    std::sort(a1.begin(), a1.end());
    std::reverse_copy(a2.begin(), a2.end(),
                      std::ostream_iterator<int>(std::cout, " "));

    std::cout << '\n';

    // ranged for loop is supported
    for(const auto& s: a3)
        std::cout << s << ' ';
}
```

Output:

```
3 2 1
a b
```

std::array::at

`reference at( size_type pos );`	(since C++11)
`const_reference at( size_type pos ) const;`	(since C++11) (until C++14)
`constexpr const_reference at( size_type pos ) const;`	(since C++14)

Returns a reference to the element at specified location `pos`, with bounds checking.

If `pos` not within the range of the container, an exception of type `std::out_of_range` is thrown.

Parameters

`pos` - position of the element to return

Return value

Reference to the requested element.

Exceptions

`std::out_of_range` if `!(pos < size())`.

Complexity

Constant.

See also

`operator[]`	access specified element (public member function)

std::array::operator[]

`reference operator[]( size_type pos );`	(since C++11)
`const_reference operator[]( size_type pos ) const;`	(since C++11) (until C++14)
`constexpr const_reference operator[]( size_type pos ) const;`	(since C++14)

Returns a reference to the element at specified location `pos`. No bounds checking is performed.

Parameters

pos - position of the element to return

Return value

Reference to the requested element.

Complexity

Constant.

Notes

Unlike `std::map::operator[]`, this operator never inserts a new element into the container.

Example

The following code uses `operator[]` to read from and write to a `std::array<int>`:

Run this code

```
#include <array>
#include <iostream>

int main()
{
    std::array<int,4> numbers {2, 4, 6, 8};

    std::cout << "Second element: " << numbers[1] << '\n';

    numbers[0] = 5;

    std::cout << "All numbers:";
    for (auto i : numbers) {
        std::cout << ' ' << i;
    }
    std::cout << '\n';
}
```

Output:

```
Second element: 4
All numbers: 5 4 6 8
```

See also

`at`	access specified element with bounds checking (public member function)

std::array::front

`reference front();`	(since C++11)
`const_reference front() const;`	(since C++11) (until C++14)
`constexpr const_reference front() const;`	(since C++14)

Returns a reference to the first element in the container.

Calling `front` on an empty container is undefined.

Parameters

(none)

Return value

reference to the first element

Complexity

Constant

Notes

For a container `c`, the expression `c.front()` is equivalent to `*c.begin()`.

Example

The following code uses `front` to display the first element of a `std::array<char>`:

Run this code

```
#include <array>
#include <iostream>

int main()
{
    std::array<char> letters {'o', 'm', 'g', 'w', 't', 'f'};

    if (!letters.empty()) {
        std::cout << "The first character is: " << letters.front() << '\n';
    }
}
```

Output:

```
The first character is o
```

See also

back	access the last element (public member function)

std::array::back

`reference back();`	(since C++11)
`const_reference back() const;`	(since C++11) (until C++14)
`constexpr const_reference back() const;`	(since C++14)

Returns reference to the last element in the container.

Calling `back` on an empty container is undefined.

Parameters

(none)

Return value

Reference to the last element.

Complexity

Constant.

Notes

For a container `c`, the expression `return c.back();` is equivalent to `{ auto tmp = c.end(); --tmp; return *tmp; }`

Example

The following code uses `back` to display the last element of a `std::array<char>`:

Run this code

```
#include <array>
#include <iostream>

int main()
{
    std::array<char, 6> letters {'o', 'm', 'g', 'w', 't', 'f'};
    if (!letters.empty()) {
        std::cout << "The last character is: " << letters.back() << '\n';
    }
}
```

Output:

```
The last character is f
```

See also

`front`	access the first element (public member function)

std::array::data

`T* data();`	(since C++11)
`const T* data() const;`	(since C++11)

Returns pointer to the underlying array serving as element storage. The pointer is such that range `[data(); data() + size())` is always a valid range, even if the container is empty.

Parameters

(none)

Return value

Pointer to the underlying element storage. For non-empty containers, returns `&front()`

Complexity

Constant.

Exceptions

`noexcept` specification: `noexcept`

See also

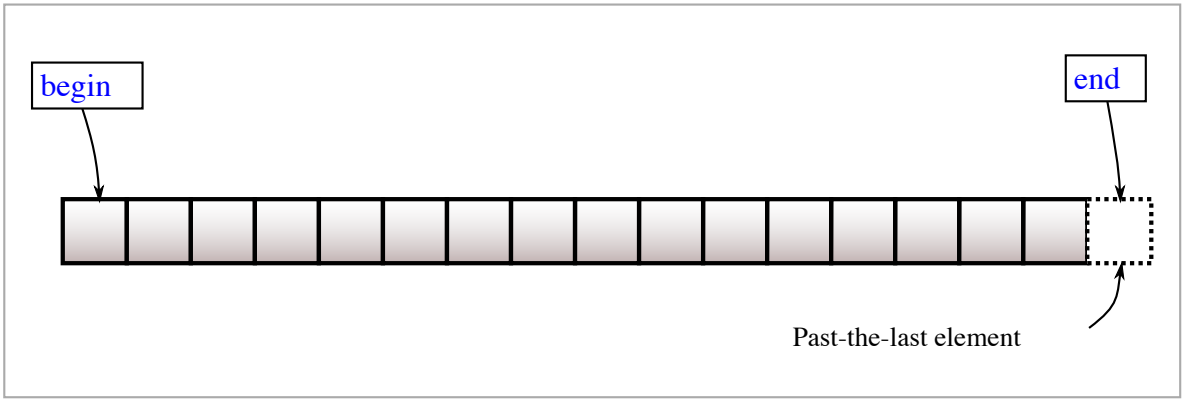
`front`	access the first element (public member function)
`back`	access the last element (public member function)

std::array::begin, std::array::cbegin

```
iterator begin();                    (since C++11)
const_iterator begin() const;        (since C++11)
const_iterator cbegin() const;       (since C++11)
```

Returns an iterator to the first element of the container.

If the container is empty, the returned iterator will be equal to `end()`.



Parameters

(none)

Return value

Iterator to the first element

Exceptions

`noexcept` specification: `noexcept`

Complexity

Constant

Example

This section is incomplete
Reason: no example

See also

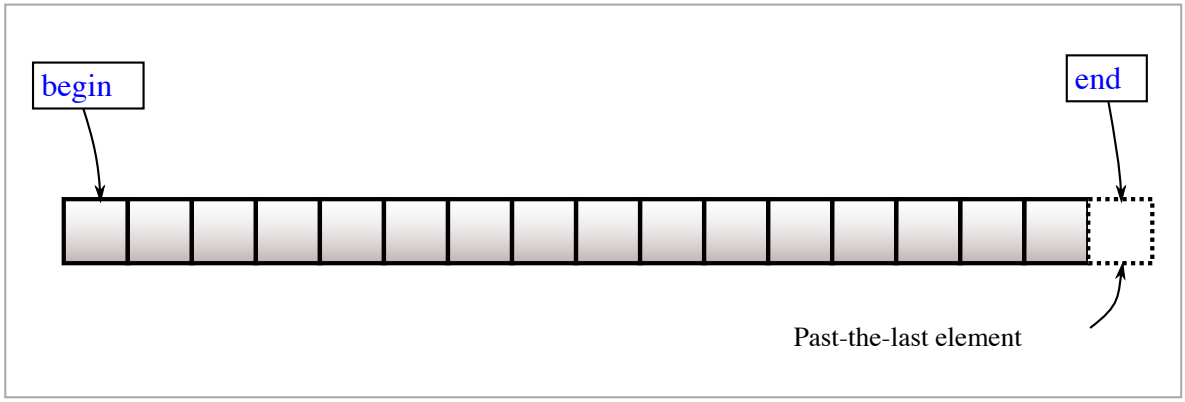
end **cend**	returns an iterator to the end (public member function)

std::array::end, std::array::cend

```
iterator end();                    (since C++11)
const_iterator end() const;        (since C++11)
const_iterator cend() const;       (since C++11)
```

Returns an iterator to the element following the last element of the container.

This element acts as a placeholder; attempting to access it results in undefined behavior.



Parameters

(none)

Return value

Iterator to the element following the last element.

Exceptions

`noexcept` specification: `noexcept`

Complexity

Constant.

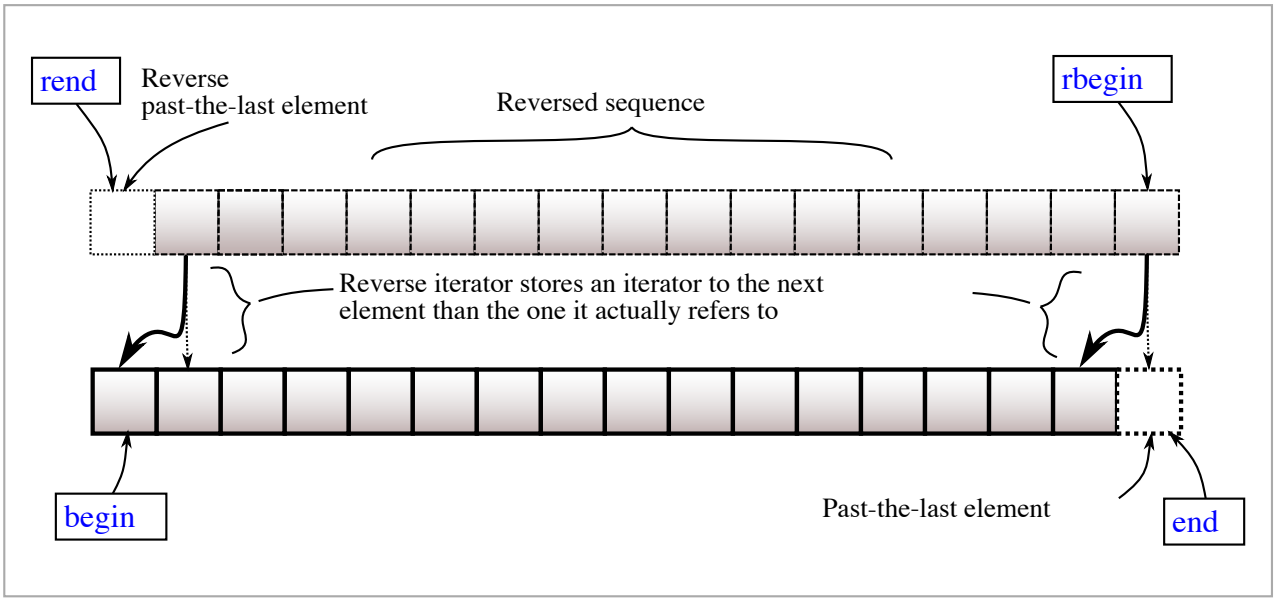
See also

begin **cbegin**	returns an iterator to the beginning (public member function)

std::array::rbegin, std::array::crbegin

```
reverse_iterator rbegin();                    (since C++11)
const_reverse_iterator rbegin() const;        (since C++11)
const_reverse_iterator crbegin() const;       (since C++11)
```

Returns a reverse iterator to the first element of the reversed container. It corresponds to the last element of the non-reversed container.



Parameters

(none)

Return value

Reverse iterator to the first element.

Exceptions

`noexcept` specification: `noexcept`

Complexity

Constant.

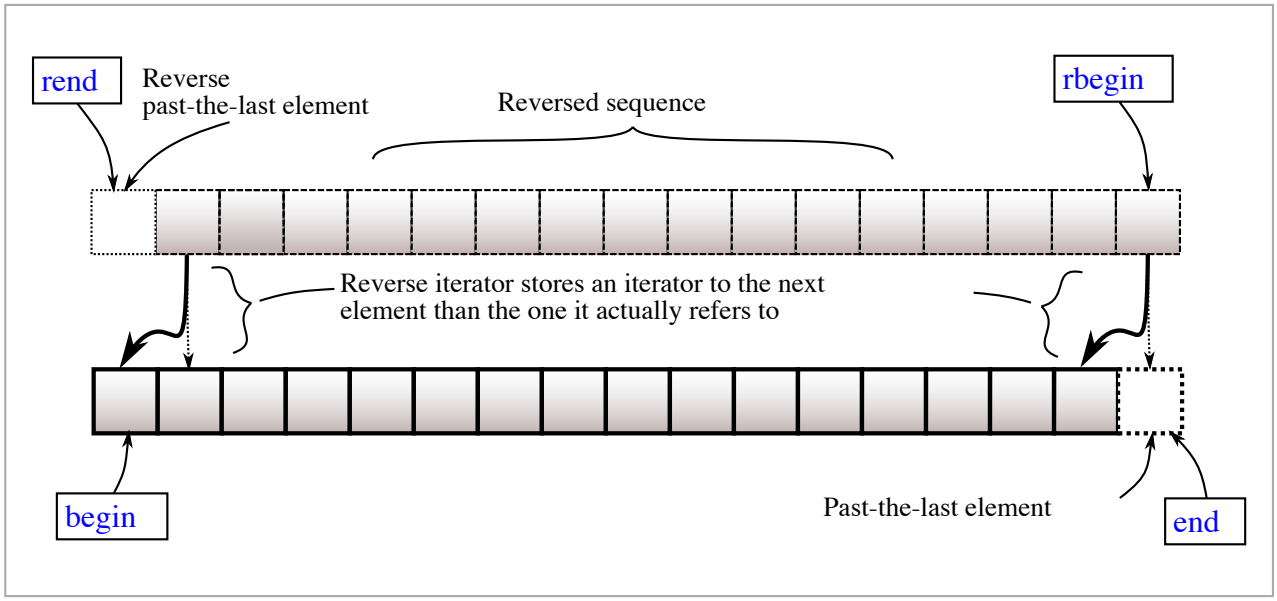
See also

rend **crend**	returns a reverse iterator to the end (public member function)

std::array::rend, std::array::crend

```
reverse_iterator rend();                    (since C++11)
const_reverse_iterator rend() const;        (since C++11)
const_reverse_iterator crend() const;       (since C++11)
```

Returns a reverse iterator to the element following the last element of the reversed container. It corresponds to the element preceding the first element of the non-reversed container. This element acts as a placeholder, attempting to access it results in undefined behavior.



Parameters

(none)

Return value

Reverse iterator to the element following the last element.

Exceptions

`noexcept` specification: `noexcept`

Complexity

Constant.

See also

rbegin	returns a reverse iterator to the beginning
crbegin	(public member function)

std::array::empty

`constexpr bool empty();`	(since C++11) (until C++14)
`constexpr bool empty() const;`	(since C++14)

Checks if the container has no elements, i.e. whether `begin() == end()`.

Parameters

(none)

Return value

`true` if the container is empty, `false` otherwise

Exceptions

`noexcept` specification: `noexcept`

Complexity

Constant.

Example

The following code uses `empty` to check if a `std::array` contains any elements:

Run this code

```
#include <array>
#include <iostream>

int main()
{
    std::array<int, 4> numbers {3, 1, 4, 1};
    std::array<int, 0> no_numbers;

    std::cout << "numbers.empty(): " << numbers.empty() << '\n';
    std::cout << "no_numbers.empty(): " << no_numbers.empty() << '\n';
}
```

Output:

```
numbers.empty(): 0
no_numbers.empty(): 1
```

See also

`size`	returns the number of elements (public member function)

std::array::size

`constexpr size_type size();`	(since C++11) (until C++14)
`constexpr size_type size() const;`	(since C++14)

Returns the number of elements in the container, i.e. `std::distance(begin(), end())`.

Parameters

(none)

Return value

The number of elements in the container.

Exceptions

`noexcept` specification: `noexcept`

Complexity

Constant.

Example

The following code uses `size` to display the number of elements in a `std::array`:

Run this code

```
#include <array>
#include <iostream>

int main()
{
    std::array<int, 4> nums {1, 3, 5, 7};

    std::cout << "nums contains " << nums.size() << " elements.\n";
}
```

Output:

```
nums contains 4 elements.
```

See also

empty	checks whether the container is empty (public member function)
max_size	returns the maximum possible number of elements (public member function)

std::array::max_size

`constexpr size_type max_size();`	(since C++11) (until C++14)
`constexpr size_type max_size() const;`	(since C++14)

Returns the maximum number of elements the container is able to hold due to system or library implementation limitations, i.e. `std::distance(begin(), end())` for the largest container.

Parameters

(none)

Return value

Maximum number of elements.

Exceptions

`noexcept` specification: `noexcept`

Complexity

Constant.

Notes

Because each `std::array<T, N>` is a fixed-size container, the value returned by `max_size` equals `N` (which is also the value returned by `size`)

Example

Run this code

```
#include <iostream>
#include <array>

int main()
{
    std::array<char, 10> s;
    std::cout << "Maximum size of a 'array' is " << s.max_size() << "\n";
}
```

Possible output:

```
Maximum size of a 'array' is 10
```

See also

`size`	returns the number of elements (public member function)

std::array::fill

```
void fill( const T& value );    (since C++11)
```

Assigns the given value `value` to all elements in the container.

Parameters

`value` - the value to assign to the elements

Return value

(none)

Complexity

Linear in the size of the container

std::array::swap

```
void swap( array& other );    (since C++11)
```

Exchanges the contents of the container with those of `other`. Does not cause iterators and references to associate with the other container.

Parameters

other - container to exchange the contents with

Return value

(none)

Exceptions

`noexcept` specification:

```
noexcept(noexcept(std::swap(declval<T&>(), declval<T&>())))
```

For zero-sized arrays, `noexcept` specification:

```
noexcept(noexcept(true))
```

Complexity

Linear in size of the container.

See also

std::swap(std::array) (C++11)	specializes the `std::swap` algorithm (function template)

std::get(std::array)

```
template< size_t I, class T, size_t N >
constexpr T& get( array<T,N>& a );                          (1)  (since C++11)

template< size_t I, class T, size_t N >
constexpr T&& get( array<T,N>&& a );                        (2)  (since C++11)

template< size_t I, class T, size_t N >
constexpr const T& get( const array<T,N>& a );              (3)  (since C++11)
```

Extracts the `Ith` element element from the array.

`I` must be an integer value in range `[0, N)`. This is enforced at compile time as opposed to `at()` or `operator[]`.

Parameters

a - array whose contents to extract

Return value

1) Reference to the `Ith` element of `a`.

2) Rvalue reference to the `Ith` element of `a`, unless the element is of lvalue reference type, in which case lvalue reference is returned.

3) Const reference to the `Ith` element of `a`.

Exceptions

`noexcept` specification: `noexcept`

Notes

The overloads are marked as `constexpr` since C++14.

Example

Run this code

```
#include <iostream>
#include <array>

int main()
{
    std::array<int, 3> arr;

    // set values:
    std::get<0>(arr) = 1;
    std::get<1>(arr) = 2;
    std::get<2>(arr) = 3;

    // get values:
    std::cout << "(" << std::get<0>(arr) << ", " << std::get<1>(arr)
              << ", " << std::get<2>(arr) << ")\n";
}
```

Output:

```
(1, 2, 3)
```

See also

operator[]	access specified element (public member function)
at	access specified element with bounds checking (public member function)
std::get(std::tuple)	tuple accesses specified element (function template)
std::get(std::pair) (C++11)	accesses an element of a `pair` (function template)

std::swap(std::array)

```
template< class T, std::size_t N >
void swap( array<T,N>& lhs,           (since C++11)
           array<T,N>& rhs );
```

Specializes the `std::swap` algorithm for `std::array`. Swaps the contents of `lhs` and `rhs`. Calls `lhs.swap(rhs)`.

Parameters

`lhs, rhs` - containers whose contents to swap

Return value

(none)

Complexity

Linear in size of the container.

Exceptions

(since C++17)

`noexcept` specification:

```
noexcept(noexcept(lhs.swap(rhs)))
```

See also

`swap`	swaps the contents (public member function)

std::tuple_size(std::array)

Defined in header <array>

```
template< class T, size_t N >
class tuple_size< array<T, N> > :
    public integral_constant<size_t, N>
{ };
```

(1) (since C++11)

Provides access to the number of elements in an `std::array` as a compile-time constant expression.

Inherited from std::integral_constant

Member constants

value [static]	`N`, the number of elements in the array (public static member constant)

Member functions

operator std::size_t	converts the object to `std::size_t`, returns `value` (public member function)
operator() (C++14)	returns `value` (public member function)

Member types

Type	Definition
value_type	`std::size_t`
type	`std::integral_constant<std::size_t, value>`

Example

Run this code

```
#include <iostream>
#include <array>

template<class T>
void test(T t)
{
    int a[std::tuple_size<T>::value]; // can be used at compile time
    std::cout << std::tuple_size<T>::value << '\n';
}

int main()
{
    std::array<float, 3> arr;
    test(arr);
}
```

Output:

```
3
```

See also

tuple_size	obtains the size of `tuple` at compile time (class template specialization)

std::tuple_element<std::array>

Defined in header <array>

```
template< std::size_t I, class T, std::size_t N >
struct tuple_element<I, array<T, N> >;
```
(since C++11)

Provides compile-time indexed access to the type of the elements of the array using tuple-like interface

Member types

Member type	Definition
type	the type of elements of the array

Possible implementation

```
template<std::size_t I, typename T>
  struct tuple_element;

template<std::size_t I, typename T, std::size_t N>
  struct tuple_element<I, std::array<T,N> >
  {
     using type = T;
  };
```

Example

This section is incomplete
Reason: no example

See also

tuple_element	obtains the type of the specified element (class template specialization)
std::tuple_element<std::pair> (C++11)	obtains the type of the elements of pair (class template specialization)

operator==,!=,<,<=,>,>=(std::array)

```
template< class T, std::size_t N >
bool operator==( const array<T,N>& lhs,      (1)
                 const array<T,N>& rhs );
template< class T, std::size_t N >
bool operator!=( const array<T,N>& lhs,      (2)
                 const array<T,N>& rhs );
template< class T, std::size_t N >
bool operator<( const array<T,N>& lhs,       (3)
                const array<T,N>& rhs );
template< class T, std::size_t N >
bool operator<=( const array<T,N>& lhs,      (4)
                 const array<T,N>& rhs );
template< class T, std::size_t N >
bool operator>( const array<T,N>& lhs,       (5)
                const array<T,N>& rhs );
template< class T, std::size_t N >
bool operator>=( const array<T,N>& lhs,      (6)
                 const array<T,N>& rhs );
```

Compares the contents of two containers.

1-2) Checks if the contents of `lhs` and `rhs` are equal, that is, whether each element in `lhs` compares equal with the element in `rhs` at the same position.

3-6) Compares the contents of `lhs` and `rhs` lexicographically. The comparison is performed by a function equivalent to `std::lexicographical_compare`.

Parameters

`lhs, rhs` - containers whose contents to compare

- `T` must meet the requirements of `EqualityComparable` in order to use overloads (1-2).
- `T` must meet the requirements of `LessThanComparable` in order to use overloads (3-6). The ordering relation must establish total order.

Return value

1) `true` if the contents of the containers are equal, `false` otherwise
2) `true` if the contents of the containers are not equal, `false` otherwise
3) `true` if the contents of the `lhs` are lexicographically *less* than the contents of `rhs`, `false` otherwise
4) `true` if the contents of the `lhs` are lexicographically *less* than or *equal* the contents of `rhs`, `false` otherwise
5) `true` if the contents of the `lhs` are lexicographically *greater* than the contents of `rhs`, `false` otherwise
6) `true` if the contents of the `lhs` are lexicographically *greater* than or *equal* the contents of `rhs`, `false` otherwise

Complexity

Linear in the size of the container