**Array**

template < class T, size\_t N > class array;

defines the fixed-size array container class

# Array class

1. fixed-size sequence containers
2. they hold a specific number of elements ordered in a strict linear sequence
3. Internally, an array does not keep any data other than the elements it contains (not even its size)
4. as efficient in terms of storage size as an ordinary array
5. arrays have a fixed size and do not manage the allocation of its elements through an allocator
6. an aggregate type encapsulating a fixed-size array of elements
7. they cannot be expanded or contracted dynamically
8. Zero-sized arrays are valid, but they should not be dereferenced
9. swapping two array containers is a linear operation that involves swapping all the elements in the ranges individually
10. swapping two array containers is a less efficient operation
11. this allows the iterators to elements in both containers to keep their original container association
12. can be treated as tuple objects
13. The <array> header overloads the get function to access the elements of the array

# Container properties

## Sequence:

* Elements in sequence containers are ordered in a strict linear sequence.
* Individual elements are accessed by their position in this 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.

# Member functions

## Iterators

[**begin**](http://www.cplusplus.com/reference/array/array/begin/)

Return iterator to beginning (public member function )

[**end**](http://www.cplusplus.com/reference/array/array/end/)

Return iterator to end (public member function )

[**rbegin**](http://www.cplusplus.com/reference/array/array/rbegin/)

Return reverse iterator to reverse beginning (public member function )

[**rend**](http://www.cplusplus.com/reference/array/array/rend/)

Return reverse iterator to reverse end (public member function )

[**cbegin**](http://www.cplusplus.com/reference/array/array/cbegin/)

Return const\_iterator to beginning (public member function )

[**cend**](http://www.cplusplus.com/reference/array/array/cend/)

Return const\_iterator to end (public member function )

[**crbegin**](http://www.cplusplus.com/reference/array/array/crbegin/)

Return const\_reverse\_iterator to reverse beginning (public member function )

[**crend**](http://www.cplusplus.com/reference/array/array/crend/)

Return const\_reverse\_iterator to reverse end (public member function )

## Capacity

[**size**](http://www.cplusplus.com/reference/array/array/size/)

Return size (public member function )

[**max\_size**](http://www.cplusplus.com/reference/array/array/max_size/)

Return maximum size (public member function )

[**empty**](http://www.cplusplus.com/reference/array/array/empty/)

Test whether array is empty (public member function )

## Element access

[**operator[]**](http://www.cplusplus.com/reference/array/array/operator%5b%5d/)

Access element (public member function )

[**at**](http://www.cplusplus.com/reference/array/array/at/)

Access element (public member function )

[**front**](http://www.cplusplus.com/reference/array/array/front/)

Access first element (public member function )

[**back**](http://www.cplusplus.com/reference/array/array/back/)

Access last element (public member function )

[**data**](http://www.cplusplus.com/reference/array/array/data/)

Get pointer to data (public member function )

## Modifiers

[**fill**](http://www.cplusplus.com/reference/array/array/fill/)

Fill array with value (public member function )

[**swap**](http://www.cplusplus.com/reference/array/array/swap/)

Swap content (public member function )

# Non-member function overloads

[**get (array)**](http://www.cplusplus.com/reference/array/array/get/)

Get element (tuple interface) (function template )

[**relational operators (array)**](http://www.cplusplus.com/reference/array/array/operators/)

Relational operators for array (function template )

# Non-member class specializations

[**tuple\_element<array>**](http://www.cplusplus.com/reference/array/array/tuple_element/)

Tuple element type for array (class template specialization )

[**tuple\_size<array>**](http://www.cplusplus.com/reference/array/array/tuple_size/)

Tuple size traits for array (class template specialization )

# Iterators

## begin

Return iterator to beginning (public member function )

|  |
| --- |
| **iterator begin() noexcept;** |
| **const\_iterator begin() const noexcept;** |

* Return iterator to beginning
* Returns an iterator pointing to the first element in the array container
* this function returns a random access iterator pointing to it (unlike member array::front, which returns a reference to the first element)
* In zero-sized arrays, returns array::end, but the returned iterator should not be dereferenced

**Complexity**: Constant

**Return**: An iterator to the beginning of the sequence

**Data races**: Concurrently accessing or modifying different elements is safe

**Exception safety**: never throws exceptions, copy construction or assignment of the returned iterator is also guaranteed to never throw

## End

Return iterator to end (public member function )

|  |
| --- |
| **iterator end() noexcept;** |
| **const\_iterator end() const noexcept;** |

* Return iterator to end
* Returns an iterator to the past-the-end element
* past-the-end element: element following the last element of the container, It does not point to any element, and thus shall not be dereferenced
* In zero-sized arrays, this function returns the same as array::begin

**Complexity**: Constant

**Return**: An iterator to the beginning of the sequence

**Data races**: Concurrently accessing or modifying different elements is safe

**Exception safety**: never throws exceptions, copy construction or assignment of the returned iterator is also guaranteed to never throw

#include <iostream>

#include <array>

int main ()

{

std::array<int,5> myarray = { 2, 16, 77, 34, 50 };

std::cout << "myarray contains:";

for ( auto it = myarray.begin(); it != myarray.end(); ++it )

std::cout << ' ' << \*it;

std::cout << '\n';

return 0;

}

Output:

myarray contains: 2 16 77 34 50

# Capacity

## size

Return size (public member function )

**constexpr size\_type size() noexcept;**

* Return size
* Returns the number of elements in the array container
* equal to the second template parameter used to instantiate the array template class (N)
* returns the size of the array in terms of number of elements (not size in bytes)

**Complexity**: Constant

**Return**: number of elements contained in the array object (This is a compile-time constant expression constexpr)

**Data races**: concurrently accessing or modifying them is safe

**Exception safety**: this member function never throws exceptions

std::array<int,5> myints;

std::cout << "size of myints: " << myints.size() << std::endl;

std::cout << "sizeof(myints): " << sizeof(myints) << std::endl;

output:

size of myints: 5

sizeof(myints): 20

## max\_size

Return maximum size (public member function )

**constexpr size\_type max\_size() noexcept;**

* Return maximum size
* Returns the maximum number of elements the container is able to hold due to system or library implementation limitations
* max\_size of an array object, is just like its size, is always equal to the second template parameter

**Complexity**: Constant

**Return**: maximum number of elements the object can hold as content

**Data races**: concurrently accessing or modifying them is safe

**Exception safety**: this member function never throws exceptions

std::array<int,10> myints;

std::cout << "size of myints: " << myints.size() << '\n';

std::cout << "max\_size of myints: " << myints.max\_size() << '\n';

Output:

size of myints: 10

max\_size of myints: 10

## empty

Test whether array is empty (public member function )

**constexpr bool empty() noexcept;**

* Test whether array is empty i.e. whether begin() == end()
* does not modify the content of the array in any way

**Complexity**: Constant

**Return**: true if the array size is 0, false otherwise.

**Data races**: concurrently accessing or modifying them is safe

**Exception safety**: this member function never throws exceptions

std::array<int,0> first;

std::array<int,5> second;

std::cout << "first " << (first.empty() ? "is empty" : "is not empty") << '\n';

std::cout << "second " << (second.empty() ? "is empty" : "is not empty") << '\n';

Output:

first is empty

second is not empty

# Element access

## operator[]

Access element (public member function )

|  |
| --- |
| **reference operator[] (size\_type n);** |
| **const\_reference operator[] (size\_type n) const;** |

* Access element
* Returns a reference to the element at position n in the array container
* do not checks the array bounds
* reference returned can be used to access or modify elements

**Complexity**: Constant

**Return**: element at the specified position in the array

**Data races**: Concurrently accessing or modifying different elements is safe

**Exception safety**: If the container size > n, the function never throws exceptions (no-throw guarantee)

Otherwise, it causes undefined behavior

example:

int main () {

std::array<int,10> myarray;

unsigned int i;

for (i=0; i<10; i++) // assign some values:

myarray[i]=i;

// print content

std::cout << "myarray contains:";

for (i=0; i<10; i++)

std::cout << ' ' << myarray[i];

std::cout << '\n';

return 0;

}

Output:

myarray contains: 0 1 2 3 4 5 6 7 8 9

## at

Access element (public member function )

|  |
| --- |
| **reference at ( size\_type n );** |
| **const\_reference at ( size\_type n ) const;** |

* Access element
* Returns a reference to the element at position n in the array
* checks whether n is within the bounds of valid elements in the container
* throws out\_of\_range exception if n >= size of array
* reference returned can be used to access or modify elements

**Complexity**: Constant

**Return**: element at the specified position in the array

**Data races**: Concurrently accessing or modifying different elements is safe

**Exception safety**: It throws out\_of\_range if n is out of bounds

if an exception is thrown, there are no changes in the container (gauranted)

example:

int main ()

{

std::array<int,10> myarray;

// assign some values:

for (int i=0; i<10; i++) myarray.at(i) = i+1;

// print content:

std::cout << "myarray contains:";

for (int i=0; i<10; i++)

std::cout << ' ' << myarray.at(i);

std::cout << '\n';

return 0;

}

Output:

myarray contains: 1 2 3 4 5 6 7 8 9 10

## front

Access first element (public member function )

|  |
| --- |
| **reference front();** |
| **const\_reference front() const;** |

* Access first element
* Returns a reference to the first element in the array container
* **c.front() is equivalent to \*c.begin()**
* this function returns a direct reference to the first element in the array (array::begin returns an iterator )
* Calling this function on an empty container causes undefined behavior
* reference returned can be used to access or modify elements

**Complexity**: Constant

**Return**: A reference to the first element in the array

**Data races**: Concurrently accessing or modifying different elements is safe

**Exception safety**: If the container is not empty, the function never throws exceptions (no-throw guarantee)

Otherwise, it causes undefined behavior

## back

Access last element (public member function )

|  |
| --- |
| **reference back();** |
| **const\_reference back() const;** |

* Access last element
* Returns a reference to the last element in the array container
* return c.back(); is equivalent to { auto tmp = c.end(); --tmp; return \*tmp; }
* this function returns a direct reference to the last element in the array (array::end returns an iterator)
* Calling this function on an empty container causes undefined behavior
* reference returned can be used to access or modify elements

**Complexity**: Constant

**Return**: A reference to the last element in the array

**Data races**: Concurrently accessing or modifying different elements is safe

**Exception safety**: If the container is not empty, the function never throws exceptions (no-throw guarantee)

Otherwise, it causes undefined behavior

example:

int main ()

{

std::array<int,3> myarray = {5, 19, 77};

std::cout << "front is: " << myarray.front() << std::endl; // 5

std::cout << "back is: " << myarray.back() << std::endl; // 77

myarray.back() = 50;

std::cout << "myarray now contains:";

for ( int& x : myarray ) std::cout << ' ' << x;

std::cout << '\n';

return 0;

}

Output:

front is: 5

back is: 77

myarray now contains: 5 19 50

## data

Get pointer to data (public member function )

|  |
| --- |
| **value\_type\* data() noexcept;** |
| **const value\_type\* data() const noexcept;** |

* Get pointer to data
* Returns pointer to the underlying array serving as element storage
* For non-empty containers, the pointer = address of the first element
* the pointer retrieved can be offset to access any element in the array
* pointer is such that range [data(); data() + size()] is always a valid range, even if the container is empty
* data() is not dereferenceable in the case container is empty

**Complexity**: Constant

**Return**: Pointer to the data contained by the array object

**Data races**: Concurrently accessing or modifying different elements is safe

**Exception safety**: this member function never throws exceptions (No-throw guarantee)

example:

int main ()

{

const char\* cstr = "Test string";

std::array<char,12> charray;

std::memcpy (charray.data(),cstr,12);

std::cout << charray.data() << '\n';

return 0;

}

Output:

Test string

# Modifiers

## fill

Fill array with value (public member function )

**void fill (const value\_type& val);**

* Fill array with value
* Sets val as the value for all the elements in the array object

**Complexity**: Linear in the size of the container

**Return**: none

**Data races**: All contained elements are modified

**Exception safety**: It throws if the assignment of any element throws

if an exception is thrown, the container is in a valid state (Basic guarantee)

example:

int main () {

std::array<int,6> myarray;

myarray.fill(5);

std::cout << "myarray contains:";

for ( int& x : myarray) { std::cout << ' ' << x; }

std::cout << '\n';

return 0;

}

Output:

myarray contains: 5 5 5 5 5 5

## swap

Swap content (public member function )

**void swap (array& x) noexcept(noexcept(swap(declval<value\_type&>(),**

**declval<value\_type&>())));**

* Swap content
* Exchanges the content of the array by the content of x, which is another array object of the same type (including the same size)
* operates in linear time by performing as many individual swap operations between the individual elements as their size
* iterators, references and pointers is not changed, but the elements they still refer to have the swapped values

**Complexity**: Linear in the size of the container

**Return**: none

**Data races**: Both the container and x are modified

**Exception safety**: If the non-member specialization of swap for the type of the elements is non-throwing, the function never throws exceptions (no-throw guarantee).

Otherwise, the container is guaranteed to end in a valid state (basic guarantee).

# Non-member function overloads

## get (array)

Get element (tuple interface) (function template )

|  |
| --- |
| **//(since C++11)** |
| **template <size\_t I, class T, size\_t N> T& get (array<T,N>& arr) noexcept;** |
| **template <size\_t I, class T, size\_t N> T&& get (array<T,N>&& arr) noexcept;** |
| **template <size\_t I, class T, size\_t N> const T& get (const array<T,N>& arr) noexcept;** |
| **//(since C++17)** |
| **template <size\_t I, class T, size\_t N> const T&& get (const array<T,N>&& arr) noexcept;** |

* Get element (tuple interface)
* 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[]

**Complexity**: Constant

**Return**: A reference to the Ith element of a

**Data races**: Concurrently accessing or modifying other elements of arr is safe

**Exception safety**: No-throw guarantee: this function never throws exceptions

## Relational Operators (array)

|  |  |
| --- | --- |
| (1) | template <class T, size\_T N>  bool operator== ( const array<T,N>& lhs, const array<T,N>& rhs ); |
| (2) | **template <class T, size\_T N>**  **bool operator!= ( const array<T,N>& lhs, const array<T,N>& rhs );** |
| (3) | **template <class T, size\_T N>**  **bool operator< ( const array<T,N>& lhs, const array<T,N>& rhs );** |
| (4) | **template <class T, size\_T N>**  **bool operator<= ( const array<T,N>& lhs, const array<T,N>& rhs );** |
| (5) | **template <class T, size\_T N>**  **bool operator> ( const array<T,N>& lhs, const array<T,N>& rhs );** |
| (6) | **template <class T, size\_T N>**  **bool operator>= ( const array<T,N>& lhs, 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.

**Complexity**: Linear in size of the container.

**Return**:

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

**Data races**: Up to all of the elements contained in both lhs and rhs may be accessed.

**Exception safety**:

If the type of the elements supports the appropriate operation with no-throw guarantee, the function never throws exceptions (no-throw guarantee).

In any case, the function cannot modify its arguments.

#include <iostream>

#include <array>

int main ()

{

std::array<int,5> a = {10, 20, 30, 40, 50};

std::array<int,5> b = {10, 20, 30, 40, 50};

std::array<int,5> c = {50, 40, 30, 20, 10};

if (a==b) std::cout << "a and b are equal\n";

if (b!=c) std::cout << "b and c are not equal\n";

if (b<c) std::cout << "b is less than c\n";

if (c>b) std::cout << "c is greater than b\n";

if (a<=b) std::cout << "a is less than or equal to b\n";

if (a>=b) std::cout << "a is greater than or equal to b\n";

return 0;

}

Output:

a and b are equal

b and c are not equal

b is less than c

c is greater than b

a is less than or equal to b

a is greater than or equal to b