std::**vector**

Defined in header <vector>

`template<`
`    class T,`
`    class Allocator = std::allocator<T>`
`> class vector;`
	(1)	

`namespace pmr {`
`    template <class T>`
`    using vector = std::vector<T, std::pmr::polymorphic_allocator<T>>;`
`}`
	(2)	(since C++17)

1) `std::vector` is a sequence container that encapsulates dynamic size arrays.

2) `std::pmr::vector` is an alias template that uses a polymorphic allocator.

The elements are stored contiguously, which means that elements can be accessed not only through iterators, but also using offsets to regular pointers to elements. This means that a pointer to an element of a vector may be passed to any function that expects a pointer to an element of an array.

The storage of the vector is handled automatically, being expanded and contracted as needed. Vectors usually occupy more space than static arrays, because more memory is allocated to handle future growth. This way a vector does not need to reallocate each time an element is inserted, but only when the additional memory is exhausted. The total amount of allocated memory can be queried using `capacity()` function. Extra memory can be returned to the system via a call to `shrink_to_fit()`. (since C++11)

Reallocations are usually costly operations in terms of performance. The `reserve()` function can be used to eliminate reallocations if the number of elements is known beforehand.

The complexity (efficiency) of common operations on vectors is as follows:

- Random access - constant $\mathcal{O}(1)$
- Insertion or removal of elements at the end - amortized constant $\mathcal{O}(1)$
- Insertion or removal of elements - linear in the distance to the end of the vector $\mathcal{O}(n)$

`std::vector` (for `T` other than `bool`) meets the requirements of *Container*, *AllocatorAwareContainer*, *SequenceContainer* , *ContiguousContainer* (since C++17) and *ReversibleContainer*.

Member functions of `std::vector` are `constexpr`: it is possible to create and use `std::vector` objects in the evaluation of a constant expression. However, `std::vector` objects generally cannot be `constexpr`, because any dynamically allocated storage must be released in the same evaluation of constant expression.	(since C++20)

Template parameters

T - The type of the elements.

T must meet the requirements of *CopyAssignable* and *CopyConstructible*.	(until C++11)
The requirements that are imposed on the elements depend on the actual operations performed on the container. Generally, it is required that element type is a complete type and meets the requirements of *Erasable*, but many member functions impose stricter requirements.	(since C++11) (until C++17)
The requirements that are imposed on the elements depend on the actual operations performed on the container. Generally, it is required that element type meets the requirements of *Erasable*, but many member functions impose stricter requirements. This container (but not its members) can be instantiated with an incomplete element type if the allocator satisfies the allocator completeness requirements.	(since C++17)

`Allocator` - An allocator that is used to acquire/release memory and to construct/destroy the elements in that memory. The type must meet the requirements of *Allocator*. The behavior is undefined (until C++20) The program is ill-formed (since C++20) if `Allocator::value_type` is not the same as `T`.

Specializations

The standard library provides a specialization of std::vector for the type bool, which may be optimized for space efficiency.

vector<bool>	space-efficient dynamic bitset (class template specialization)

Iterator invalidation

Operations	Invalidated
All read only operations	Never
swap, std::swap	end()
clear, operator=, assign	Always
reserve, shrink_to_fit	If the vector changed capacity, all of them. If not, none.
erase	Erased elements and all elements after them (including end())
push_back, emplace_back	If the vector changed capacity, all of them. If not, only end().
insert, emplace	If the vector changed capacity, all of them. If not, only those at or after the insertion point (including end()).
resize	If the vector changed capacity, all of them. If not, only end() and any elements erased.
pop_back	The element erased and end().

Member types

Member type	Definition
value_type	T
allocator_type	Allocator
size_type	Unsigned integer type (usually std::size_t)
difference_type	Signed integer type (usually std::ptrdiff_t)
reference	value_type&
const_reference	const value_type&
pointer	Allocator::pointer (until C++11) std::allocator_traits<Allocator>::pointer (since C++11)
const_pointer	Allocator::const_pointer (until C++11) std::allocator_traits<Allocator>::const_pointer (since C++11)
iterator	*LegacyRandomAccessIterator* to value_type
const_iterator	*LegacyRandomAccessIterator* to const value_type
reverse_iterator	std::reverse_iterator<iterator>
const_reverse_iterator	std::reverse_iterator<const_iterator>

Member functions

(constructor)	constructs the vector (public member function)
(destructor)	destructs the vector (public member function)
operator=	assigns values to the container (public member function)
assign	assigns values to the container (public member function)
get_allocator	returns the associated allocator (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` (C++11)	direct access to the underlying array (public member function)

Iterators

`begin` **`cbegin`** (C++11)	returns an iterator to the beginning (public member function)
`end` **`cend`** (C++11)	returns an iterator to the end (public member function)
`rbegin` **`crbegin`** (C++11)	returns a reverse iterator to the beginning (public member function)
`rend` **`crend`** (C++11)	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)
`reserve`	reserves storage (public member function)
`capacity`	returns the number of elements that can be held in currently allocated storage (public member function)
`shrink_to_fit` (C++11)	reduces memory usage by freeing unused memory (public member function)

Modifiers

`clear`	clears the contents (public member function)
`insert`	inserts elements (public member function)
`emplace` (C++11)	constructs element in-place (public member function)
`erase`	erases elements (public member function)
`push_back`	adds an element to the end (public member function)
`emplace_back` (C++11)	constructs an element in-place at the end (public member function)
`pop_back`	removes the last element (public member function)
`resize`	changes the number of elements stored (public member function)
`swap`	swaps the contents (public member function)

Non-member functions

`operator==` **`operator!=`** (removed in C++20) **`operator<`** (removed in C++20) **`operator<=`** (removed in C++20) **`operator>`** (removed in C++20) **`operator>=`** (removed in C++20) **`operator<=>`** (C++20)	lexicographically compares the values in the vector (function template)

std::swap(std::vector)	specializes the std::swap algorithm (function template)
erase(std::vector) **erase_if**(std::vector) (C++20)	Erases all elements satisfying specific criteria (function template)

Deduction guides(since C++17)

Example

Run this code

```
#include <iostream>
#include <vector>

int main()
{
    // Create a vector containing integers
    std::vector<int> v = { 7, 5, 16, 8 };

    // Add two more integers to vector
    v.push_back(25);
    v.push_back(13);

    // Print out the vector
    std::cout << "v = { ";
    for (int n : v) {
        std::cout << n << ", ";
    }
    std::cout << "}; \n";
}
```

Output:

```
v = { 7, 5, 16, 8, 25, 13, };
```

Defect reports

The following behavior-changing defect reports were applied retroactively to previously published C++ standards.

DR	Applied to	Behavior as published	Correct behavior
LWG 69 (https://cplusplus.github.io/LWG/issue69)	C++98	contiguity of the storage for elements of vector was not required	required

Retrieved from "https://en.cppreference.com/mwiki/index.php?title=cpp/container/vector&oldid=128957"