



vector<T, Alloc>

Containers

Category: containers

Type

Component type: type

Description

A vector is a [Sequence](#) that supports random access to elements, constant time insertion and removal of elements at the end, and linear time insertion and removal of elements at the beginning or in the middle. The number of elements in a vector may vary dynamically; memory management is automatic. Vector is the simplest of the STL container classes, and in many cases the most efficient.

Example

```
vector<int> V;  
V.insert(V.begin(), 3);  
assert(V.size() == 1 && V.capacity() >= 1 && V[0] == 3);
```

Definition

Defined in the standard header [vector](#), and in the nonstandard backward-compatibility header [vector.h](#).

Template parameters

| Parameter | Description | Default |
|-----------|---|-----------------------|
| T | The vector's value type: the type of object that is stored in the vector. | |
| Alloc | The vector's allocator, used for all internal memory management. | alloc |

Model of

[Random Access Container](#), [Back Insertion Sequence](#).

Type requirements

None, except for those imposed by the requirements of [Random Access Container](#) and [Back Insertion Sequence](#).

Public base classes

None.

Members

| Member | Where defined | Description |
|---------------------------------------|--------------------------------------|--|
| value_type | Container | The type of object, T , stored in the vector. |
| pointer | Container | Pointer to T . |
| reference | Container | Reference to T |
| const_reference | Container | Const reference to T |
| size_type | Container | An unsigned integral type. |
| difference_type | Container | A signed integral type. |
| iterator | Container | Iterator used to iterate through a vector. |
| const_iterator | Container | Const iterator used to iterate through a vector. |
| reverse_iterator | Reversible Container | Iterator used to iterate backwards through a vector. |
| const_reverse_iterator | Reversible Container | Const iterator used to iterate backwards through a vector. |
| iterator begin() | Container | Returns an iterator pointing to the beginning of the vector. |
| iterator end() | Container | Returns an iterator pointing to the end of the vector. |
| const_iterator begin() const | Container | Returns a const_iterator pointing to the beginning of the vector. |
| const_iterator end() const | Container | Returns a const_iterator pointing to the end of the vector. |
| reverse_iterator rbegin() | Reversible Container | Returns a reverse_iterator pointing to the beginning of the reversed vector. |
| reverse_iterator rend() | Reversible Container | Returns a reverse_iterator pointing to the end of the reversed vector. |
| const_reverse_iterator rbegin() const | Reversible Container | Returns a const_reverse_iterator pointing to the beginning of the reversed vector. |
| const_reverse_iterator rend() const | Reversible Container | Returns a const_reverse_iterator pointing to the end of the reversed vector. |
| size_type size() const | Container | Returns the size of the vector. |
| size_type max_size() const | Container | Returns the largest possible size of the vector. |
| size_type capacity() const | vector | See below. |

| | | |
|--|---|---|
| <code>bool empty() const</code> | Container | true if the vector's size is 0. |
| <code>reference operator[](size_type n)</code> | Random Access Container | Returns the n'th element. |
| <code>const_reference operator[](size_type n) const</code> | Random Access Container | Returns the n'th element. |
| <code>vector()</code> | Container | Creates an empty vector. |
| <code>vector(size_type n)</code> | Sequence | Creates a vector with n elements. |
| <code>vector(size_type n, const T& t)</code> | Sequence | Creates a vector with n copies of t. |
| <code>vector(const vector&)</code> | Container | The copy constructor. |
| <code>template <class InputIterator></code> <code>vector(InputIterator, InputIterator)</code> [1] | Sequence | Creates a vector with a copy of a range. |
| <code>~vector()</code> | Container | The destructor. |
| <code>vector& operator=(const vector&)</code> | Container | The assignment operator |
| <code>void reserve(size_t)</code> | vector | See below. |
| <code>reference front()</code> | Sequence | Returns the first element. |
| <code>const_reference front() const</code> | Sequence | Returns the first element. |
| <code>reference back()</code> | Back Insertion Sequence | Returns the last element. |
| <code>const_reference back() const</code> | Back Insertion Sequence | Returns the last element. |
| <code>void push_back(const T&)</code> | Back Insertion Sequence | Inserts a new element at the end. |
| <code>void pop_back()</code> | Back Insertion Sequence | Removes the last element. |
| <code>void swap(vector&)</code> | Container | Swaps the contents of two vectors. |
| <code>iterator insert(iterator pos,</code> <code> const T& x)</code> | Sequence | Inserts x before pos. |
| <code>template <class InputIterator></code> <code>void insert(iterator pos,</code> <code> InputIterator f, InputIterator l)</code> [1] | Sequence | Inserts the range [first, last) before pos. |
| <code>void insert(iterator pos,</code> <code> size_type n, const T& x)</code> | Sequence | Inserts n copies of x before pos. |
| <code>iterator erase(iterator pos)</code> | Sequence | Erases the element at position pos. |
| <code>iterator erase(iterator first, iterator last)</code> | Sequence | Erases the range [first, last) |
| <code>void clear()</code> | Sequence | Erases all of the elements. |
| <code>void resize(n, t = T())</code> | | |

| | | |
|--|-----------------------------------|---|
| | Sequence | Inserts or erases elements at the end such that the size becomes n. |
| <code>bool operator==(const vector&, const vector&)</code> | Forward Container | Tests two vectors for equality. This is a global function, not a member function. |
| <code>bool operator<(const vector&, const vector&)</code> | Forward Container | Lexicographical comparison. This is a global function, not a member function. |

New members

These members are not defined in the [Random Access Container](#) and [Back Insertion Sequence](#) requirements, but are specific to vector.

| Member | Description |
|---|---|
| <code>size_type capacity() const</code> | Number of elements for which memory has been allocated. <code>capacity()</code> is always greater than or equal to <code>size()</code> . [2] [3] |
| <code>void reserve(size_type n)</code> | If n is less than or equal to <code>capacity()</code> , this call has no effect. Otherwise, it is a request for allocation of additional memory. If the request is successful, then <code>capacity()</code> is greater than or equal to n; otherwise, <code>capacity()</code> is unchanged. In either case, <code>size()</code> is unchanged. [2] [4] |

Notes

[1] This member function relies on *member template* functions, which at present (early 1998) are not supported by all compilers. If your compiler supports member templates, you can call this function with any type of [input iterator](#). If your compiler does not yet support member templates, though, then the arguments must be of type `const value_type*`.

[2] Memory will be reallocated automatically if more than `capacity() - size()` elements are inserted into the vector. Reallocation does not change `size()`, nor does it change the values of any elements of the vector. It does, however, increase `capacity()`, and it invalidates [\[5\]](#) any iterators that point into the vector.

[3] When it is necessary to increase `capacity()`, vector usually increases it by a factor of two. It is crucial that the amount of growth is proportional to the current `capacity()`, rather than a fixed constant: in the former case inserting a series of elements into a vector is a linear time operation, and in the latter case it is quadratic.

[4] `Reserve()` causes a reallocation manually. The main reason for using `reserve()` is efficiency: if you know the capacity to which your vector must eventually grow, then it is usually more efficient to allocate that memory all at once rather than relying on the automatic reallocation scheme. The other reason for using `reserve()` is so that you can control the invalidation of iterators. [\[5\]](#)

[5] A vector's iterators are invalidated when its memory is reallocated. Additionally, inserting or deleting an element in the middle of a vector invalidates all iterators that point to elements following the insertion or deletion point. It follows that you can prevent a vector's iterators from being invalidated if you use `reserve()` to preallocate as much memory as the vector will ever use, and if all insertions and deletions are at the vector's end.

See also

[deque](#), [list](#), [slist](#)

[STL Main Page](#)

[Contact Us](#) | [Site Map](#) | [Trademarks](#) | [Privacy](#) | Using this site means you accept its [Terms of Use](#)

Copyright © 2009 - 2014 Silicon Graphics International. All rights reserved.