class template

<vector>

**std::vector**

template < class T, class Alloc = allocator<T> > class vector; // generic template

Vector

Vectors are sequence containers representing arrays that can change in size.  
  
Just like arrays, vectors use contiguous storage locations for their elements, which means that their elements can also be accessed using offsets on regular pointers to its elements, and just as efficiently as in arrays. But unlike arrays, their size can change dynamically, with their storage being handled automatically by the container.  
  
Internally, vectors use a dynamically allocated array to store their elements. This array may need to be reallocated in order to grow in size when new elements are inserted, which implies allocating a new array and moving all elements to it. This is a relatively expensive task in terms of processing time, and thus, vectors do not reallocate each time an element is added to the container.  
  
Instead, vector containers may allocate some extra storage to accommodate for possible growth, and thus the container may have an actual [capacity](http://www.cplusplus.com/vector::capacity) greater than the storage strictly needed to contain its elements (i.e., its [size](http://www.cplusplus.com/vector::size)). Libraries can implement different strategies for growth to balance between memory usage and reallocations, but in any case, reallocations should only happen at logarithmically growing intervals of [size](http://www.cplusplus.com/vector::size) so that the insertion of individual elements at the end of the vector can be provided with *amortized constant time* complexity (see [push\_back](http://www.cplusplus.com/vector::push_back)).  
  
Therefore, compared to arrays, vectors consume more memory in exchange for the ability to manage storage and grow dynamically in an efficient way.  
  
Compared to the other dynamic sequence containers ([deques](http://www.cplusplus.com/deque), [lists](http://www.cplusplus.com/list) and [forward\_lists](http://www.cplusplus.com/forward_list)), vectors are very efficient accessing its elements (just like arrays) and relatively efficient adding or removing elements from its [end](http://www.cplusplus.com/vector::end). For operations that involve inserting or removing elements at positions other than the end, they perform worse than the others, and have less consistent iterators and references than [lists](http://www.cplusplus.com/list) and [forward\_lists](http://www.cplusplus.com/forward_list).

**Container properties**

Sequence

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

Dynamic array

Allows direct access to any element in the sequence, even through pointer arithmetics, and provides relatively fast addition/removal of elements at the end of the sequence.

Allocator-aware

The container uses an allocator object to dynamically handle its storage needs.

**Template parameters**

T

Type of the elements.  
Only if T [is guaranteed to not throw while moving](http://www.cplusplus.com/is_nothrow_move_constructible), implementations can optimize to move elements instead of copying them during reallocations.  
Aliased as member type vector::value\_type.

Alloc

Type of the allocator object used to define the storage allocation model. By default, the [allocator](http://www.cplusplus.com/allocator) class template is used, which defines the simplest memory allocation model and is value-independent.  
Aliased as member type vector::allocator\_type.

**Member types**

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

|  |  |  |
| --- | --- | --- |
| **member type** | **definition** | **notes** |
| value\_type | The first template parameter (T) |  |
| allocator\_type | The second template parameter (Alloc) | defaults to: [allocator](http://www.cplusplus.com/allocator)<value\_type> |
| reference | allocator\_type::reference | for the default [allocator](http://www.cplusplus.com/allocator): value\_type& |
| const\_reference | allocator\_type::const\_reference | for the default [allocator](http://www.cplusplus.com/allocator): const value\_type& |
| pointer | allocator\_type::pointer | for the default [allocator](http://www.cplusplus.com/allocator): value\_type\* |
| const\_pointer | allocator\_type::const\_pointer | for the default [allocator](http://www.cplusplus.com/allocator): const value\_type\* |
| iterator | a [random access iterator](http://www.cplusplus.com/RandomAccessIterator) to value\_type | convertible to const\_iterator |
| const\_iterator | a [random access iterator](http://www.cplusplus.com/RandomAccessIterator) to const value\_type |  |
| reverse\_iterator | [reverse\_iterator](http://www.cplusplus.com/reverse_iterator)<iterator> |  |
| const\_reverse\_iterator | [reverse\_iterator](http://www.cplusplus.com/reverse_iterator)<const\_iterator> |  |
| difference\_type | a signed integral type, identical to: iterator\_traits<iterator>::difference\_type | usually the same as [ptrdiff\_t](http://www.cplusplus.com/ptrdiff_t) |
| size\_type | an unsigned integral type that can represent any non-negative value of difference\_type | usually the same as [size\_t](http://www.cplusplus.com/size_t) |

**Member functions**

[**(constructor)**](http://www.cplusplus.com/reference/vector/vector/vector/)

Construct vector (public member function )

[**(destructor)**](http://www.cplusplus.com/reference/vector/vector/%7Evector/)

Vector destructor (public member function )

[**operator=**](http://www.cplusplus.com/reference/vector/vector/operator=/)

Assign content (public member function )

**Iterators**:

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

Return iterator to beginning (public member function )

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

Return iterator to end (public member function )

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

Return reverse iterator to reverse beginning (public member function )

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

Return reverse iterator to reverse end (public member function )

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

Return const\_iterator to beginning (public member function )

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

Return const\_iterator to end (public member function )

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

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

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

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

**Capacity**:

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

Return size (public member function )

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

Return maximum size (public member function )

[**resize**](http://www.cplusplus.com/reference/vector/vector/resize/)

Change size (public member function )

[**capacity**](http://www.cplusplus.com/reference/vector/vector/capacity/)

Return size of allocated storage capacity (public member function )

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

Test whether vector is empty (public member function )

[**reserve**](http://www.cplusplus.com/reference/vector/vector/reserve/)

Request a change in capacity (public member function )

[**shrink\_to\_fit**](http://www.cplusplus.com/reference/vector/vector/shrink_to_fit/)

Shrink to fit (public member function )

**Element access**:

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

Access element (public member function )

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

Access element (public member function )

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

Access first element (public member function )

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

Access last element (public member function )

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

Access data (public member function )

**Modifiers**:

[**assign**](http://www.cplusplus.com/reference/vector/vector/assign/)

Assign vector content (public member function )

[**push\_back**](http://www.cplusplus.com/reference/vector/vector/push_back/)

Add element at the end (public member function )

[**pop\_back**](http://www.cplusplus.com/reference/vector/vector/pop_back/)

Delete last element (public member function )

[**insert**](http://www.cplusplus.com/reference/vector/vector/insert/)

Insert elements (public member function )

[**erase**](http://www.cplusplus.com/reference/vector/vector/erase/)

Erase elements (public member function )

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

Swap content (public member function )

[**clear**](http://www.cplusplus.com/reference/vector/vector/clear/)

Clear content (public member function )

[**emplace**](http://www.cplusplus.com/reference/vector/vector/emplace/)

Construct and insert element (public member function )

[**emplace\_back**](http://www.cplusplus.com/reference/vector/vector/emplace_back/)

Construct and insert element at the end (public member function )

**Allocator**:

[**get\_allocator**](http://www.cplusplus.com/reference/vector/vector/get_allocator/)

Get allocator (public member function )

**Non-member function overloads**

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

Relational operators for vector (function template )

[**swap**](http://www.cplusplus.com/reference/vector/vector/swap-free/)

Exchange contents of vectors (function template )

**Template specializations**

[**vector<bool>**](http://www.cplusplus.com/reference/vector/vector-bool/)

Vector of bool (class template specialization )

**[C++](http://www.cplusplus.com/)**

* [Information](http://www.cplusplus.com/info/)
* [Tutorials](http://www.cplusplus.com/doc/)
* [Reference](http://www.cplusplus.com/reference/)
* [Articles](http://www.cplusplus.com/articles/)
* [Forum](http://www.cplusplus.com/forum/)

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

* **[C library:](http://www.cplusplus.com/reference/clibrary/)**
  + [<cassert> (assert.h)](http://www.cplusplus.com/reference/cassert/)
  + [<cctype> (ctype.h)](http://www.cplusplus.com/reference/cctype/)
  + [<cerrno> (errno.h)](http://www.cplusplus.com/reference/cerrno/)
  + [<cfenv> (fenv.h)](http://www.cplusplus.com/reference/cfenv/)
  + [<cfloat> (float.h)](http://www.cplusplus.com/reference/cfloat/)
  + [<cinttypes> (inttypes.h)](http://www.cplusplus.com/reference/cinttypes/)
  + [<ciso646> (iso646.h)](http://www.cplusplus.com/reference/ciso646/)
  + [<climits> (limits.h)](http://www.cplusplus.com/reference/climits/)
  + [<clocale> (locale.h)](http://www.cplusplus.com/reference/clocale/)
  + [<cmath> (math.h)](http://www.cplusplus.com/reference/cmath/)
  + [<csetjmp> (setjmp.h)](http://www.cplusplus.com/reference/csetjmp/)
  + [<csignal> (signal.h)](http://www.cplusplus.com/reference/csignal/)
  + [<cstdarg> (stdarg.h)](http://www.cplusplus.com/reference/cstdarg/)
  + [<cstdbool> (stdbool.h)](http://www.cplusplus.com/reference/cstdbool/)
  + [<cstddef> (stddef.h)](http://www.cplusplus.com/reference/cstddef/)
  + [<cstdint> (stdint.h)](http://www.cplusplus.com/reference/cstdint/)
  + [<cstdio> (stdio.h)](http://www.cplusplus.com/reference/cstdio/)
  + [<cstdlib> (stdlib.h)](http://www.cplusplus.com/reference/cstdlib/)
  + [<cstring> (string.h)](http://www.cplusplus.com/reference/cstring/)
  + [<ctgmath> (tgmath.h)](http://www.cplusplus.com/reference/ctgmath/)
  + [<ctime> (time.h)](http://www.cplusplus.com/reference/ctime/)
  + [<cuchar> (uchar.h)](http://www.cplusplus.com/reference/cuchar/)
  + [<cwchar> (wchar.h)](http://www.cplusplus.com/reference/cwchar/)
  + [<cwctype> (wctype.h)](http://www.cplusplus.com/reference/cwctype/)
* **[Containers:](http://www.cplusplus.com/reference/stl/)**
  + [<array>](http://www.cplusplus.com/reference/array/)
  + [<deque>](http://www.cplusplus.com/reference/deque/)
  + [<forward\_list>](http://www.cplusplus.com/reference/forward_list/)
  + [<list>](http://www.cplusplus.com/reference/list/)
  + [<map>](http://www.cplusplus.com/reference/map/)
  + [<queue>](http://www.cplusplus.com/reference/queue/)
  + [<set>](http://www.cplusplus.com/reference/set/)
  + [<stack>](http://www.cplusplus.com/reference/stack/)
  + [<unordered\_map>](http://www.cplusplus.com/reference/unordered_map/)
  + [<unordered\_set>](http://www.cplusplus.com/reference/unordered_set/)
  + [<vector>](http://www.cplusplus.com/reference/vector/)
* **[Input/Output:](http://www.cplusplus.com/reference/iolibrary/)**
  + [<fstream>](http://www.cplusplus.com/reference/fstream/)
  + [<iomanip>](http://www.cplusplus.com/reference/iomanip/)
  + [<ios>](http://www.cplusplus.com/reference/ios/)
  + [<iosfwd>](http://www.cplusplus.com/reference/iosfwd/)
  + [<iostream>](http://www.cplusplus.com/reference/iostream/)
  + [<istream>](http://www.cplusplus.com/reference/istream/)
  + [<ostream>](http://www.cplusplus.com/reference/ostream/)
  + [<sstream>](http://www.cplusplus.com/reference/sstream/)
  + [<streambuf>](http://www.cplusplus.com/reference/streambuf/)
* **[Multi-threading:](http://www.cplusplus.com/reference/multithreading/)**
  + [<atomic>](http://www.cplusplus.com/reference/atomic/)
  + [<condition\_variable>](http://www.cplusplus.com/reference/condition_variable/)
  + [<future>](http://www.cplusplus.com/reference/future/)
  + [<mutex>](http://www.cplusplus.com/reference/mutex/)
  + [<thread>](http://www.cplusplus.com/reference/thread/)
* **[Other:](http://www.cplusplus.com/reference/std/)**
  + [<algorithm>](http://www.cplusplus.com/reference/algorithm/)
  + [<bitset>](http://www.cplusplus.com/reference/bitset/)
  + [<chrono>](http://www.cplusplus.com/reference/chrono/)
  + [<codecvt>](http://www.cplusplus.com/reference/codecvt/)
  + [<complex>](http://www.cplusplus.com/reference/complex/)
  + [<exception>](http://www.cplusplus.com/reference/exception/)
  + [<functional>](http://www.cplusplus.com/reference/functional/)
  + [<initializer\_list>](http://www.cplusplus.com/reference/initializer_list/)
  + [<iterator>](http://www.cplusplus.com/reference/iterator/)
  + [<limits>](http://www.cplusplus.com/reference/limits/)
  + [<locale>](http://www.cplusplus.com/reference/locale/)
  + [<memory>](http://www.cplusplus.com/reference/memory/)
  + [<new>](http://www.cplusplus.com/reference/new/)
  + [<numeric>](http://www.cplusplus.com/reference/numeric/)
  + [<random>](http://www.cplusplus.com/reference/random/)
  + [<ratio>](http://www.cplusplus.com/reference/ratio/)
  + [<regex>](http://www.cplusplus.com/reference/regex/)
  + [<stdexcept>](http://www.cplusplus.com/reference/stdexcept/)
  + [<string>](http://www.cplusplus.com/reference/string/)
  + [<system\_error>](http://www.cplusplus.com/reference/system_error/)
  + [<tuple>](http://www.cplusplus.com/reference/tuple/)
  + [<typeindex>](http://www.cplusplus.com/reference/typeindex/)
  + [<typeinfo>](http://www.cplusplus.com/reference/typeinfo/)
  + [<type\_traits>](http://www.cplusplus.com/reference/type_traits/)
  + [<utility>](http://www.cplusplus.com/reference/utility/)
  + [<valarray>](http://www.cplusplus.com/reference/valarray/)

**[<vector>](http://www.cplusplus.com/reference/vector/)**

* [vector](http://www.cplusplus.com/reference/vector/vector/)
* [vector<bool>](http://www.cplusplus.com/reference/vector/vector-bool/)

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

* [vector::vector](http://www.cplusplus.com/reference/vector/vector/vector/)
* [vector::~vector](http://www.cplusplus.com/reference/vector/vector/%7Evector/)
* **member functions:**
  + [vector::assign](http://www.cplusplus.com/reference/vector/vector/assign/)
  + [vector::at](http://www.cplusplus.com/reference/vector/vector/at/)
  + [vector::back](http://www.cplusplus.com/reference/vector/vector/back/)
  + [vector::begin](http://www.cplusplus.com/reference/vector/vector/begin/)
  + [vector::capacity](http://www.cplusplus.com/reference/vector/vector/capacity/)
  + [vector::cbegin](http://www.cplusplus.com/reference/vector/vector/cbegin/)
  + [vector::cend](http://www.cplusplus.com/reference/vector/vector/cend/)
  + [vector::clear](http://www.cplusplus.com/reference/vector/vector/clear/)
  + [vector::crbegin](http://www.cplusplus.com/reference/vector/vector/crbegin/)
  + [vector::crend](http://www.cplusplus.com/reference/vector/vector/crend/)
  + [vector::data](http://www.cplusplus.com/reference/vector/vector/data/)
  + [vector::emplace](http://www.cplusplus.com/reference/vector/vector/emplace/)
  + [vector::emplace\_back](http://www.cplusplus.com/reference/vector/vector/emplace_back/)
  + [vector::empty](http://www.cplusplus.com/reference/vector/vector/empty/)
  + [vector::end](http://www.cplusplus.com/reference/vector/vector/end/)
  + [vector::erase](http://www.cplusplus.com/reference/vector/vector/erase/)
  + [vector::front](http://www.cplusplus.com/reference/vector/vector/front/)
  + [vector::get\_allocator](http://www.cplusplus.com/reference/vector/vector/get_allocator/)
  + [vector::insert](http://www.cplusplus.com/reference/vector/vector/insert/)
  + [vector::max\_size](http://www.cplusplus.com/reference/vector/vector/max_size/)
  + [vector::operator=](http://www.cplusplus.com/reference/vector/vector/operator=/)
  + [vector::operator[]](http://www.cplusplus.com/reference/vector/vector/operator%5b%5d/)
  + [vector::pop\_back](http://www.cplusplus.com/reference/vector/vector/pop_back/)
  + [vector::push\_back](http://www.cplusplus.com/reference/vector/vector/push_back/)
  + [vector::rbegin](http://www.cplusplus.com/reference/vector/vector/rbegin/)
  + [vector::rend](http://www.cplusplus.com/reference/vector/vector/rend/)
  + [vector::reserve](http://www.cplusplus.com/reference/vector/vector/reserve/)
  + [vector::resize](http://www.cplusplus.com/reference/vector/vector/resize/)
  + [vector::shrink\_to\_fit](http://www.cplusplus.com/reference/vector/vector/shrink_to_fit/)
  + [vector::size](http://www.cplusplus.com/reference/vector/vector/size/)
  + [vector::swap](http://www.cplusplus.com/reference/vector/vector/swap/)
* **non-member overloads:**
  + [relational operators (vector)](http://www.cplusplus.com/reference/vector/vector/operators/)
  + [swap (vector)](http://www.cplusplus.com/reference/vector/vector/swap-free/)

public member function

<vector>

# std::[vector](http://www.cplusplus.com/reference/vector/vector/)::vector

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

|  |  |
| --- | --- |
| **default (1)** | explicit vector (const allocator\_type& alloc = allocator\_type()); |
| **fill (2)** | explicit vector (size\_type n, const value\_type& val = value\_type(),  const allocator\_type& alloc = allocator\_type()); |
| **range (3)** | template <class InputIterator>  vector (InputIterator first, InputIterator last,  const allocator\_type& alloc = allocator\_type()); |
| **copy (4)** | vector (const vector& x); |

Construct vector

Constructs a [vector](http://www.cplusplus.com/vector), initializing its contents depending on the constructor version used:

* [C++98](javascript:switch2.select(1))
* [C++11](javascript:switch2.select(2))

(1) empty container constructor (default constructor)

Constructs an [empty](http://www.cplusplus.com/vector::empty) container, with no elements.

(2) fill constructor

Constructs a container with *n* elements. Each element is a copy of *val*.

(3) range constructor

Constructs a container with as many elements as the range [first,last), with each element constructed from its corresponding element in that range, in the same order.

(4) copy constructor

Constructs a container with a copy of each of the elements in *x*, in the same order.

The container keeps an internal copy of *alloc*, which is used to allocate storage throughout its lifetime.  
The copy constructor *(4)* creates a container that keeps and uses a copy of *x*'s allocator.  
  
The storage for the elements is allocated using this [internal allocator](http://www.cplusplus.com/vector::get_allocator).

### Parameters

alloc

Allocator object.  
The container keeps and uses an internal copy of this allocator.  
Member type allocator\_type is the internal allocator type used by the container, defined in [vector](http://www.cplusplus.com/vector) as an alias of its second template parameter (Alloc).  
If allocator\_type is an instantiation of the default [allocator](http://www.cplusplus.com/allocator) (which has no state), this is not relevant.

n

Initial container size (i.e., the number of elements in the container at construction).  
Member type size\_type is an unsigned integral type.

val

Value to fill the container with. Each of the *n* elements in the container will be initialized to a copy of this value.  
Member type value\_type is the type of the elements in the container, defined in [vector](http://www.cplusplus.com/vector) as an alias of its first template parameter (T).

first, last

[Input iterators](http://www.cplusplus.com/InputIterator) to the initial and final positions in a range. The range used is [first,last), which includes all the elements between *first* and *last*, including the element pointed by *first* but not the element pointed by *last*.  
The function template argument InputIterator shall be an [input iterator](http://www.cplusplus.com/InputIterator) type that points to elements of a type from which value\_type objects can be constructed.

x

Another [vector](http://www.cplusplus.com/vector) object of the same type (with the same class template arguments T and Alloc), whose contents are either copied or acquired.

il

An [initializer\_list](http://www.cplusplus.com/initializer_list) object.  
These objects are automatically constructed from *initializer list* declarators.  
Member type value\_type is the type of the elements in the container, defined in [vector](http://www.cplusplus.com/vector) as an alias of its first template parameter (T).

### Example

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 | // constructing vectors  #include <iostream>  #include <vector>  int main ()  {  unsigned int i;  // constructors used in the same order as described above:  std::vector<int> first; // empty vector of ints  std::vector<int> second (4,100); // four ints with value 100  std::vector<int> third (second.begin(),second.end()); // iterating through second  std::vector<int> fourth (third); // a copy of third  // the iterator constructor can also be used to construct from arrays:  int myints[] = {16,2,77,29};  std::vector<int> fifth (myints, myints + sizeof(myints) / sizeof(int) );  std::cout << "The contents of fifth are:";  for (std::vector<int>::iterator it = fifth.begin(); it != fifth.end(); ++it)  std::cout << ' ' << \*it;  std::cout << '\n';  return 0;  } |

Output:

|  |
| --- |
| The contents of fifth are: 16 2 77 29 |

### Complexity

Constant for the *default constructor (1)*, and for the *move constructors (5)* (unless *alloc* is different from *x*'s allocator).  
For all other cases, linear in the resulting container [size](http://www.cplusplus.com/vector::size).  
Additionally, if InputIterator in the *range constructor (3)* is not at least of a [forward iterator](http://www.cplusplus.com/ForwardIterator) category (i.e., it is just an [input iterator](http://www.cplusplus.com/InputIterator)), the new capacity cannot be determined beforehand and the construction incurs in additional logarithmic complexity in [size](http://www.cplusplus.com/vector::size) (reallocations while growing).

### Iterator validity

The *move constructors* (5), invalidate all iterators, pointers and references related to *x* if the elements are moved.

### Data races

All copied elements are accessed.  
The *move constructors (5)* modify *x*.

### Exception safety

**Strong guarantee:** no effects in case an exception is thrown.  
If [allocator\_traits::construct](http://www.cplusplus.com/allocator_traits::construct) is not supported with the appropriate arguments for the element constructions, or if the range specified by [first,last) is not valid, it causes *undefined behavior*.

public member function

<vector>

# std::[vector](http://www.cplusplus.com/reference/vector/vector/)::~vector

~vector();

Vector destructor

Destroys the container object.

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

This destroys all container elements, and deallocates all the storage [capacity](http://www.cplusplus.com/vector::capacity) allocated by the [vector](http://www.cplusplus.com/vector) using its [allocator](http://www.cplusplus.com/vector::get_allocator).

### Complexity

Linear in [vector::size](http://www.cplusplus.com/vector::size) (destructors).

### Iterator validity

All iterators, pointers and references are invalidated.

### Data races

The container and all its elements are modified.

### Exception safety

**No-throw guarantee:** never throws exceptions.

public member function

<vector>

# std::[vector](http://www.cplusplus.com/reference/vector/vector/)::operator=

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

|  |  |
| --- | --- |
| **copy (1)** | vector& operator= (const vector& x); |

Assign content

Assigns new contents to the container, replacing its current contents, and modifying its [size](http://www.cplusplus.com/vector::size) accordingly.

* [C++98](javascript:switch2.select(1))
* [C++11](javascript:switch2.select(2))

Copies all the elements from *x* into the container.  
  
The container preserves its [current allocator](http://www.cplusplus.com/vector::get_allocator), which is used to allocate storage in case of reallocation.

Any elements held in the container before the call are either *assigned to* or *destroyed*.

### Parameters

x

A [vector](http://www.cplusplus.com/vector) object of the same type (i.e., with the same template parameters, T and Alloc).

il

An [initializer\_list](http://www.cplusplus.com/initializer_list) object. The compiler will automatically construct such objects from *initializer list* declarators.  
Member type value\_type is the type of the elements in the container, defined in [vector](http://www.cplusplus.com/vector) as an alias of its first template parameter (T).

### Return value

\*this

### Example

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 | // vector assignment  #include <iostream>  #include <vector>  int main ()  {  std::vector<int> foo (3,0);  std::vector<int> bar (5,0);  bar = foo;  foo = std::vector<int>();  std::cout << "Size of foo: " << int(foo.size()) << '\n';  std::cout << "Size of bar: " << int(bar.size()) << '\n';  return 0;  } |

Output:

|  |
| --- |
| Size of foo: 0  Size of bar: 3 |

### Complexity

Linear in [size](http://www.cplusplus.com/vector::size).

### Iterator validity

All iterators, references and pointers related to this container before the call are invalidated.  
  
In the *move assignment*, iterators, pointers and references referring to elements in *x* are also invalidated.

### Data races

All copied elements are accessed.  
The *move assignment (2)* modifies *x*.  
The container and all its elements are modified.

### Exception safety

**Basic guarantee:** if an exception is thrown, the container is in a valid state.  
If [allocator\_traits::construct](http://www.cplusplus.com/allocator_traits::construct) is not supported with the appropriate arguments for the element constructions, or if value\_type is not [copy assignable](http://www.cplusplus.com/is_copy_assignable) (or [move assignable](http://www.cplusplus.com/is_move_assignable) for *(2)*), it causes *undefined behavior*.

public member function

<vector>

# std::[vector](http://www.cplusplus.com/reference/vector/vector/)::begin

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

iterator begin();

const\_iterator begin() const;

Return iterator to beginning

Returns an iterator pointing to the first element in the [vector](http://www.cplusplus.com/vector).  
  
Notice that, unlike member [vector::front](http://www.cplusplus.com/vector::front), which returns a reference to the first element, this function returns a [random access iterator](http://www.cplusplus.com/RandomAccessIterator) pointing to it.  
  
If the container is [empty](http://www.cplusplus.com/vector::empty), the returned iterator value shall not be dereferenced.

### Parameters

none

### Return Value

An iterator to the beginning of the sequence container.  
  
If the [vector](http://www.cplusplus.com/vector) object is const-qualified, the function returns a const\_iterator. Otherwise, it returns an iterator.  
  
Member types iterator and const\_iterator are [random access iterator](http://www.cplusplus.com/RandomAccessIterator) types (pointing to an element and to a const element, respectively).

### Example

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 | // vector::begin/end  #include <iostream>  #include <vector>  int main ()  {  std::vector<int> myvector;  for (int i=1; i<=5; i++) myvector.push\_back(i);  std::cout << "myvector contains:";  for (std::vector<int>::iterator it = myvector.begin() ; it != myvector.end(); ++it)  std::cout << ' ' << \*it;  std::cout << '\n';  return 0;  } |

Output:

|  |
| --- |
| myvector contains: 1 2 3 4 5 |

### Complexity

Constant.

### Iterator validity

No changes.

### Data races

The container is accessed (neither the const nor the non-const versions modify the container).  
No contained elements are accessed by the call, but the iterator returned can be used to access or modify elements. Concurrently accessing or modifying different elements is safe.

### Exception safety

**No-throw guarantee:** this member function never throws exceptions.  
The copy construction or assignment of the returned iterator is also guaranteed to never throw.

public member function

<vector>

# std::[vector](http://www.cplusplus.com/reference/vector/vector/)::end

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

iterator end();

const\_iterator end() const;

Return iterator to end

Returns an iterator referring to the *past-the-end* element in the [vector](http://www.cplusplus.com/vector) container.  
  
The past-the-end element is the theoretical element that would follow the last element in the [vector](http://www.cplusplus.com/vector). It does not point to any element, and thus shall not be dereferenced.  
  
Because the ranges used by functions of the standard library do not include the element pointed by their closing iterator, this function is often used in combination with [vector::begin](http://www.cplusplus.com/vector::begin) to specify a range including all the elements in the container.  
  
If the container is [empty](http://www.cplusplus.com/vector::empty), this function returns the same as [vector::begin](http://www.cplusplus.com/vector::begin).

### Parameters

none

### Return Value

An iterator to the element past the end of the sequence.  
  
If the [vector](http://www.cplusplus.com/vector) object is const-qualified, the function returns a const\_iterator. Otherwise, it returns an iterator.  
  
Member types iterator and const\_iterator are [random access iterator](http://www.cplusplus.com/RandomAccessIterator) types (pointing to an element and to a const element, respectively).

### Example

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 | // vector::begin/end  #include <iostream>  #include <vector>  int main ()  {  std::vector<int> myvector;  for (int i=1; i<=5; i++) myvector.push\_back(i);  std::cout << "myvector contains:";  for (std::vector<int>::iterator it = myvector.begin() ; it != myvector.end(); ++it)  std::cout << ' ' << \*it;  std::cout << '\n';  return 0;  } |

Output:

|  |
| --- |
| myvector contains: 1 2 3 4 5 |

### Complexity

Constant.

### Iterator validity

No changes.

### Data races

The container is accessed (neither the const nor the non-const versions modify the container).  
No contained elements are accessed by the call, but the iterator returned can be used to access or modify elements. Concurrently accessing or modifying different elements is safe.

### Exception safety

**No-throw guarantee:** this member function never throws exceptions.  
The copy construction or assignment of the returned iterator is also guaranteed to never throw.

public member function

<vector>

# std::[vector](http://www.cplusplus.com/reference/vector/vector/)::rbegin

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

reverse\_iterator rbegin();

const\_reverse\_iterator rbegin() const;

Return reverse iterator to reverse beginning

Returns a *reverse iterator* pointing to the last element in the [vector](http://www.cplusplus.com/vector) (i.e., its *reverse beginning*).  
  
Reverse iterators iterate backwards: increasing them moves them towards the beginning of the container.  
  
rbegin points to the element right before the one that would be pointed to by member [end](http://www.cplusplus.com/vector::end).  
  
Notice that unlike member [vector::back](http://www.cplusplus.com/vector::back), which returns a reference to this same element, this function returns a *reverse* [*random access iterator*](http://www.cplusplus.com/RandomAccessIterator).

### Parameters

none

### Return Value

A reverse iterator to the *reverse beginning* of the sequence container.  
  
If the [vector](http://www.cplusplus.com/vector) object is const-qualified, the function returns a const\_reverse\_iterator. Otherwise, it returns a reverse\_iterator.  
  
Member types reverse\_iterator and const\_reverse\_iterator are reverse [random access iterator](http://www.cplusplus.com/RandomAccessIterator) types (pointing to an element and to a const element, respectively). See [vector member types](http://www.cplusplus.com/vector#types).

### Example

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 | // vector::rbegin/rend  #include <iostream>  #include <vector>  int main ()  {  std::vector<int> myvector (5); // 5 default-constructed ints  std::vector<int>::reverse\_iterator rit = myvector.rbegin();  int i=0;  for (rit = myvector.rbegin(); rit!= myvector.rend(); ++rit)  \*rit = ++i;  std::cout << "myvector contains:";  for (std::vector<int>::iterator it = myvector.begin(); it != myvector.end(); ++it)  std::cout << ' ' << \*it;  std::cout << '\n';  return 0;  } |

Output:

|  |
| --- |
| myvector contains: 5 4 3 2 1 |

### Complexity

Constant.

### Iterator validity

No changes.

### Data races

The container is accessed (neither the const nor the non-const versions modify the container).  
No contained elements are accessed by the call, but the iterator returned can be used to access or modify elements. Concurrently accessing or modifying different elements is safe.

### Exception safety

**No-throw guarantee:** this member function never throws exceptions.  
The copy construction or assignment of the returned iterator is also guaranteed to never throw.

public member function

<vector>

# std::[vector](http://www.cplusplus.com/reference/vector/vector/)::rend

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

reverse\_iterator rend();

const\_reverse\_iterator rend() const;

Return reverse iterator to reverse end

Returns a *reverse iterator* pointing to the theoretical element preceding the first element in the [vector](http://www.cplusplus.com/vector) (which is considered its *reverse end*).  
  
The range between [vector::rbegin](http://www.cplusplus.com/vector::rbegin) and vector::rend contains all the elements of the [vector](http://www.cplusplus.com/vector) (in reverse order).

### Parameters

none

### Return Value

A reverse iterator to the *reverse end* of the sequence container.  
  
If the [vector](http://www.cplusplus.com/vector) object is const-qualified, the function returns a const\_reverse\_iterator. Otherwise, it returns a reverse\_iterator.  
  
Member types reverse\_iterator and const\_reverse\_iterator are reverse [random access iterator](http://www.cplusplus.com/RandomAccessIterator) types (pointing to an element and to a const element, respectively). See [vector member types](http://www.cplusplus.com/vector#types).

### Example

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 | // vector::rbegin/rend  #include <iostream>  #include <vector>  int main ()  {  std::vector<int> myvector (5); // 5 default-constructed ints  std::vector<int>::reverse\_iterator rit = myvector.rbegin();  int i=0;  for (rit = myvector.rbegin(); rit!= myvector.rend(); ++rit)  \*rit = ++i;  std::cout << "myvector contains:";  for (std::vector<int>::iterator it = myvector.begin(); it != myvector.end(); ++it)  std::cout << ' ' << \*it;  std::cout << '\n';  return 0;  } |

Output:

|  |
| --- |
| 5 4 3 2 1 |

### Complexity

Constant.

### Iterator validity

No changes.

### Data races

The container is accessed (neither the const nor the non-const versions modify the container).  
No contained elements are accessed by the call, but the iterator returned can be used to access or modify elements. Concurrently accessing or modifying different elements is safe.

### Exception safety

**No-throw guarantee:** this member function never throws exceptions.  
The copy construction or assignment of the returned iterator is also guaranteed to never throw.

public member function

<vector>

# std::[vector](http://www.cplusplus.com/reference/vector/vector/)::cbegin

const\_iterator cbegin() const noexcept;

Return const\_iterator to beginning

Returns a const\_iterator pointing to the first element in the container.  
  
A const\_iterator is an iterator that points to const content. This iterator can be increased and decreased (unless it is itself also const), just like the iterator returned by [vector::begin](http://www.cplusplus.com/vector::begin), but it cannot be used to modify the contents it points to, even if the [vector](http://www.cplusplus.com/vector) object is not itself const.  
  
If the container is [empty](http://www.cplusplus.com/vector::empty), the returned iterator value shall not be dereferenced.

### Parameters

none

### Return Value

A const\_iterator to the beginning of the sequence.  
  
Member type const\_iterator is a [random access iterator](http://www.cplusplus.com/RandomAccessIterator) type that points to a const element.

### Example

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 | // vector::cbegin/cend  #include <iostream>  #include <vector>  int main ()  {  std::vector<int> myvector = {10,20,30,40,50};  std::cout << "myvector contains:";  for (auto it = myvector.cbegin(); it != myvector.cend(); ++it)  std::cout << ' ' << \*it;  std::cout << '\n';  return 0;  } |

Output:

|  |
| --- |
| myvector contains: 10 20 30 40 50 |

### Complexity

Constant.

### Iterator validity

No changes.

### Data races

The container is accessed.  
No contained elements are accessed by the call, but the iterator returned can be used to access them. Concurrently accessing or modifying different elements is safe.

### Exception safety

**No-throw guarantee:** this member function never throws exceptions.  
The copy construction or assignment of the returned iterator is also guaranteed to never throw.

public member function

<vector>

# std::[vector](http://www.cplusplus.com/reference/vector/vector/)::cend

const\_iterator cend() const noexcept;

Return const\_iterator to end

Returns a const\_iterator pointing to the *past-the-end* element in the container.  
  
A const\_iterator is an iterator that points to const content. This iterator can be increased and decreased (unless it is itself also const), just like the iterator returned by [vector::end](http://www.cplusplus.com/vector::end), but it cannot be used to modify the contents it points to, even if the [vector](http://www.cplusplus.com/vector) object is not itself const.  
  
If the container is [empty](http://www.cplusplus.com/vector::empty), this function returns the same as [vector::cbegin](http://www.cplusplus.com/vector::cbegin).  
  
The value returned shall not be dereferenced.

### Parameters

none

### Return Value

A const\_iterator to the element past the end of the sequence.  
  
Member type const\_iterator is a [random access iterator](http://www.cplusplus.com/RandomAccessIterator) type that points to a const element.

### Example

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 | // vector::cbegin/cend  #include <iostream>  #include <vector>  int main ()  {  std::vector<int> myvector = {10,20,30,40,50};  std::cout << "myvector contains:";  for (auto it = myvector.cbegin(); it != myvector.cend(); ++it)  std::cout << ' ' << \*it;  std::cout << '\n';  return 0;  } |

Output:

|  |
| --- |
| myvector contains: 10 20 30 40 50 |

### Complexity

Constant.

### Iterator validity

No changes.

### Data races

The container is accessed.  
No contained elements are accessed by the call, but the iterator returned can be used to access them. Concurrently accessing or modifying different elements is safe.

### Exception safety

**No-throw guarantee:** this member function never throws exceptions.  
The copy construction or assignment of the returned iterator is also guaranteed to never throw.

public member function

<vector>

# std::[vector](http://www.cplusplus.com/reference/vector/vector/)::crbegin

const\_reverse\_iterator crbegin() const noexcept;

Return const\_reverse\_iterator to reverse beginning

Returns a const\_reverse\_iterator pointing to the last element in the container (i.e., its *reverse beginning*).

### Parameters

none

### Return Value

A const\_reverse\_iterator to the *reverse beginning* of the sequence.  
  
Member type const\_reverse\_iterator is a reverse [random access iterator](http://www.cplusplus.com/RandomAccessIterator) type that points to a const element (see [vector member types](http://www.cplusplus.com/vector#types)).

### Example

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 | // vector::crbegin/crend  #include <iostream>  #include <vector>  int main ()  {  std::vector<int> myvector = {1,2,3,4,5};  std::cout << "myvector backwards:";  for (auto rit = myvector.crbegin(); rit != myvector.crend(); ++rit)  std::cout << ' ' << \*rit;  std::cout << '\n';  return 0;  } |

Output:

|  |
| --- |
| myvector backwards: 5 4 3 2 1 |

### Complexity

Constant.

### Iterator validity

No changes.

### Data races

The container is accessed.  
No contained elements are accessed by the call, but the iterator returned can be used to access them. Concurrently accessing or modifying different elements is safe.

### Exception safety

**No-throw guarantee:** this member function never throws exceptions.  
The copy construction or assignment of the returned iterator is also guaranteed to never throw.

public member function

<vector>

# std::[vector](http://www.cplusplus.com/reference/vector/vector/)::crend

const\_reverse\_iterator crend() const noexcept;

Return const\_reverse\_iterator to reverse end

Returns a const\_reverse\_iterator pointing to the theoretical element preceding the first element in the container (which is considered its *reverse end*).

### Parameters

none

### Return Value

A const\_reverse\_iterator to the *reverse end* of the sequence.  
  
Member type const\_reverse\_iterator is a reverse [random access iterator](http://www.cplusplus.com/RandomAccessIterator) type that points to a const element (see [vector member types](http://www.cplusplus.com/vector#types)).

### Example

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 | // vector::crbegin/crend  #include <iostream>  #include <vector>  int main ()  {  std::vector<int> myvector = {1,2,3,4,5};  std::cout << "myvector backwards:";  for (auto rit = myvector.crbegin(); rit != myvector.crend(); ++rit)  std::cout << ' ' << \*rit;  std::cout << '\n';  return 0;  } |

Output:

|  |
| --- |
| myvector backwards: 5 4 3 2 1 |

### Complexity

Constant.

### Iterator validity

No changes.

### Data races

The container is accessed.  
No contained elements are accessed by the call, but the iterator returned can be used to access them. Concurrently accessing or modifying different elements is safe.

### Exception safety

**No-throw guarantee:** this member function never throws exceptions.  
The copy construction or assignment of the returned iterator is also guaranteed to never throw.

public member function

<vector>

# std::[vector](http://www.cplusplus.com/reference/vector/vector/)::size

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

size\_type size() const;

Return size

Returns the number of elements in the [vector](http://www.cplusplus.com/vector).  
  
This is the number of actual objects held in the [vector](http://www.cplusplus.com/vector), which is not necessarily equal to its storage [capacity](http://www.cplusplus.com/vector::capacity).

### Parameters

none

### Return Value

The number of elements in the container.  
  
Member type size\_type is an unsigned integral type.

### Example

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 | // vector::size  #include <iostream>  #include <vector>  int main ()  {  std::vector<int> myints;  std::cout << "0. size: " << myints.size() << '\n';  for (int i=0; i<10; i++) myints.push\_back(i);  std::cout << "1. size: " << myints.size() << '\n';  myints.insert (myints.end(),10,100);  std::cout << "2. size: " << myints.size() << '\n';  myints.pop\_back();  std::cout << "3. size: " << myints.size() << '\n';  return 0;  } |

Output:

|  |
| --- |
| 0. size: 0  1. size: 10  2. size: 20  3. size: 19 |

### Complexity

Constant.

### Iterator validity

No changes.

### Data races

The container is accessed.  
No contained elements are accessed: concurrently accessing or modifying them is safe.

### Exception safety

**No-throw guarantee:** this member function never throws exceptions.

public member function

<vector>

# std::[vector](http://www.cplusplus.com/reference/vector/vector/)::max\_size

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

size\_type max\_size() const;

Return maximum size

Returns the maximum number of elements that the [vector](http://www.cplusplus.com/vector) can hold.  
  
This is the maximum potential [size](http://www.cplusplus.com/vector::size) the container can reach due to known system or library implementation limitations, but the container is by no means guaranteed to be able to reach that size: it can still fail to allocate storage at any point before that size is reached.

### Parameters

none

### Return Value

The maximum number of elements a [vector](http://www.cplusplus.com/vector) container can hold as content.  
  
Member type size\_type is an unsigned integral type.

### Example

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 | // comparing size, capacity and max\_size  #include <iostream>  #include <vector>  int main ()  {  std::vector<int> myvector;  // set some content in the vector:  for (int i=0; i<100; i++) myvector.push\_back(i);  std::cout << "size: " << myvector.size() << "\n";  std::cout << "capacity: " << myvector.capacity() << "\n";  std::cout << "max\_size: " << myvector.max\_size() << "\n";  return 0;  } |

A possible output for this program could be:

|  |
| --- |
| size: 100  capacity: 128  max\_size: 1073741823 |

### Complexity

Constant.

### Iterator validity

No changes.

### Data races

The container is accessed.  
No contained elements are accessed: concurrently accessing or modifying them is safe.

### Exception safety

**No-throw guarantee:** this member function never throws exceptions.

public member function

<vector>

# std::[vector](http://www.cplusplus.com/reference/vector/vector/)::resize

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

void resize (size\_type n, value\_type val = value\_type());

Change size

Resizes the container so that it contains *n* elements.  
  
If *n* is smaller than the current container [size](http://www.cplusplus.com/vector::size), the content is reduced to its first *n* elements, removing those beyond (and destroying them).  
  
If *n* is greater than the current container [size](http://www.cplusplus.com/vector::size), the content is expanded by inserting at the end as many elements as needed to reach a size of *n*. If *val* is specified, the new elements are initialized as copies of *val*, otherwise, they are value-initialized.  
  
If *n* is also greater than the current container [capacity](http://www.cplusplus.com/vector::capacity), an automatic reallocation of the allocated storage space takes place.  
  
Notice that this function changes the actual content of the container by inserting or erasing elements from it.

### Parameters

n

New container size, expressed in number of elements.  
Member type size\_type is an unsigned integral type.

val

Object whose content is copied to the added elements in case that *n* is greater than the current container [size](http://www.cplusplus.com/vector::size).  
If not specified, the default constructor is used instead.  
Member type value\_type is the type of the elements in the container, defined in [vector](http://www.cplusplus.com/vector) as an alias of the first template parameter (T).

### Return Value

none  
  
If a reallocation happens, the storage is allocated using the container's [allocator](http://www.cplusplus.com/vector::get_allocator), which may throw exceptions on failure (for the default [allocator](http://www.cplusplus.com/allocator), bad\_alloc is thrown if the allocation request does not succeed).

### Example

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 | // resizing vector  #include <iostream>  #include <vector>  int main ()  {  std::vector<int> myvector;  // set some initial content:  for (int i=1;i<10;i++) myvector.push\_back(i);  myvector.resize(5);  myvector.resize(8,100);  myvector.resize(12);  std::cout << "myvector contains:";  for (int i=0;i<myvector.size();i++)  std::cout << ' ' << myvector[i];  std::cout << '\n';  return 0;  } |

Output:

|  |
| --- |
| myvector contains: 1 2 3 4 5 100 100 100 0 0 0 0 |

### Complexity

Linear on the number of elements inserted/erased (constructions/destructions).  
  
If a reallocation happens, the reallocation is itself up to linear in the entire [vector size](http://www.cplusplus.com/vector::size).

### Iterator validity

In case the container shrinks, all iterators, pointers and references to elements that have not been removed remain valid after the resize and refer to the same elements they were referring to before the call.  
  
If the container expands, the [end iterator](http://www.cplusplus.com/vector::end) is invalidated and, if it has to reallocate storage, all iterators, pointers and references related to this container are also invalidated.

### Data races

The container is modified.  
If a reallocation happens, all contained elements are modified.  
Otherwise, none of the elements before *n* is accessed, and concurrently accessing or modifying them is safe.

### Exception safety

If *n* is less than or equal to the [size](http://www.cplusplus.com/vector::size) of the container, the function never throws exceptions (no-throw guarantee).  
If *n* is greater and a reallocation happens, there are no changes in the container in case of exception (strong guarantee) if the type of the elements is either *copyable* or *no-throw moveable*.  
Otherwise, if an exception is thrown, the container is left with a valid state (basic guarantee).

public member function

<vector>

# std::[vector](http://www.cplusplus.com/reference/vector/vector/)::capacity

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

size\_type capacity() const;

Return size of allocated storage capacity

Returns the size of the storage space currently allocated for the [vector](http://www.cplusplus.com/vector), expressed in terms of elements.  
  
This *capacity* is not necessarily equal to the [vector size](http://www.cplusplus.com/vector::size). It can be equal or greater, with the extra space allowing to accommodate for growth without the need to reallocate on each insertion.  
  
Notice that this *capacity* does not suppose a limit on the size of the [vector](http://www.cplusplus.com/vector). When this *capacity* is exhausted and more is needed, it is automatically expanded by the container (reallocating it storage space). The theoretical limit on the [size](http://www.cplusplus.com/vector::size) of a [vector](http://www.cplusplus.com/vector) is given by member [max\_size](http://www.cplusplus.com/vector::max_size).  
  
The *capacity* of a [vector](http://www.cplusplus.com/vector) can be explicitly altered by calling member [vector::reserve](http://www.cplusplus.com/vector::reserve).

### Parameters

none

### Return Value

The size of the currently allocated storage capacity in the [vector](http://www.cplusplus.com/vector), measured in terms of the number elements it can hold.  
  
Member type size\_type is an unsigned integral type.

### Example

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 | // comparing size, capacity and max\_size  #include <iostream>  #include <vector>  int main ()  {  std::vector<int> myvector;  // set some content in the vector:  for (int i=0; i<100; i++) myvector.push\_back(i);  std::cout << "size: " << (int) myvector.size() << '\n';  std::cout << "capacity: " << (int) myvector.capacity() << '\n';  std::cout << "max\_size: " << (int) myvector.max\_size() << '\n';  return 0;  } |

A possible output for this program could be:

|  |
| --- |
| size: 100  capacity: 128  max\_size: 1073741823 |

### Complexity

Constant.

### Iterator validity

No changes.

### Data races

The container is accessed.  
No contained elements are accessed: concurrently accessing or modifying them is safe.

### Exception safety

**No-throw guarantee:** this member function never throws exceptions.

public member function

<vector>

# std::[vector](http://www.cplusplus.com/reference/vector/vector/)::empty

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

bool empty() const;

Test whether vector is empty

Returns whether the [vector](http://www.cplusplus.com/vector) is empty (i.e. whether its [size](http://www.cplusplus.com/vector::size) is 0).  
  
This function does not modify the container in any way. To clear the content of a [vector](http://www.cplusplus.com/vector), see [vector::clear](http://www.cplusplus.com/vector::clear).

### Parameters

none

### Return Value

true if the container [size](http://www.cplusplus.com/vector::size) is 0, false otherwise.

### Example

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 | // vector::empty  #include <iostream>  #include <vector>  int main ()  {  std::vector<int> myvector;  int sum (0);  for (int i=1;i<=10;i++) myvector.push\_back(i);  while (!myvector.empty())  {  sum += myvector.back();  myvector.pop\_back();  }  std::cout << "total: " << sum << '\n';  return 0;  } |

The example initializes the content of the vector to a sequence of numbers (form 1 to 10). It then pops the elements one by one until it is empty and calculates their sum.  
  
Output:

|  |
| --- |
| total: 55 |

### Complexity

Constant.

### Iterator validity

No changes.

### Data races

The container is accessed.  
No contained elements are accessed: concurrently accessing or modifying them is safe.

### Exception safety

**No-throw guarantee:** this member function never throws exceptions.

public member function

<vector>

# std::[vector](http://www.cplusplus.com/reference/vector/vector/)::reserve

void reserve (size\_type n);

Request a change in capacity

Requests that the [vector capacity](http://www.cplusplus.com/vector::capacity) be at least enough to contain *n* elements.  
  
If *n* is greater than the current [vector capacity](http://www.cplusplus.com/vector::capacity), the function causes the container to reallocate its storage increasing its [capacity](http://www.cplusplus.com/vector::capacity) to *n* (or greater).  
  
In all other cases, the function call does not cause a reallocation and the [vector capacity](http://www.cplusplus.com/vector::capacity) is not affected.  
  
This function has no effect on the [vector size](http://www.cplusplus.com/vector::size) and cannot alter its elements.

### Parameters

n

Minimum [capacity](http://www.cplusplus.com/vector::capacity) for the [vector](http://www.cplusplus.com/vector).  
Note that the resulting [vector capacity](http://www.cplusplus.com/vector::capacity) may be equal or greater than *n*.  
Member type size\_type is an unsigned integral type.

### Return Value

none  
  
If the size requested is greater than the maximum size ([vector::max\_size](http://www.cplusplus.com/vector::max_size)), a [length\_error](http://www.cplusplus.com/length_error) exception is thrown.  
  
If case of reallocation, the storage is allocated using the container's [allocator](http://www.cplusplus.com/vector::get_allocator), which may throw exceptions on failure (for the default [allocator](http://www.cplusplus.com/allocator), bad\_alloc is thrown if the allocation request does not succeed).

### Example

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 | // vector::reserve  #include <iostream>  #include <vector>  int main ()  {  std::vector<int>::size\_type sz;  std::vector<int> foo;  sz = foo.capacity();  std::cout << "making foo grow:\n";  for (int i=0; i<100; ++i) {  foo.push\_back(i);  if (sz!=foo.capacity()) {  sz = foo.capacity();  std::cout << "capacity changed: " << sz << '\n';  }  }  std::vector<int> bar;  sz = bar.capacity();  bar.reserve(100); // this is the only difference with foo above  std::cout << "making bar grow:\n";  for (int i=0; i<100; ++i) {  bar.push\_back(i);  if (sz!=bar.capacity()) {  sz = bar.capacity();  std::cout << "capacity changed: " << sz << '\n';  }  }  return 0;  } |

Possible output:

|  |
| --- |
| making foo grow:  capacity changed: 1  capacity changed: 2  capacity changed: 4  capacity changed: 8  capacity changed: 16  capacity changed: 32  capacity changed: 64  capacity changed: 128  making bar grow:  capacity changed: 100 |

### Complexity

If a reallocation happens, linear in [vector size](http://www.cplusplus.com/vector::size) at most.

### Iterator validity

If a reallocation happens, all iterators, pointers and references related to the container are invalidated.  
Otherwise, they all keep referring to the same elements they were referring to before the call.

### Data races

If a reallocation happens, the container and all its contained elements are modified.  
Otherwise, the container is accessed, but not the contained elements: concurrently accessing or modifying them is safe.

### Exception safety

If no reallocations happen or if the type of the elements has either a non-throwing move constructor or a copy constructor, there are no changes in the container in case of exception (strong guarantee).  
Otherwise, the container is guaranteed to end in a valid state (basic guarantee).  
The function throws [length\_error](http://www.cplusplus.com/length_error) if *n* is greater than [max\_size](http://www.cplusplus.com/vector::max_size).

public member function

<vector>

# std::[vector](http://www.cplusplus.com/reference/vector/vector/)::shrink\_to\_fit

void shrink\_to\_fit();

Shrink to fit

Requests the container to reduce its [capacity](http://www.cplusplus.com/vector::capacity) to fit its [size](http://www.cplusplus.com/vector::size).  
  
The request is non-binding, and the container implementation is free to optimize otherwise and leave the [vector](http://www.cplusplus.com/vector) with a [capacity](http://www.cplusplus.com/vector::capacity) greater than its [size](http://www.cplusplus.com/vector::size).  
  
This may cause a reallocation, but has no effect on the [vector size](http://www.cplusplus.com/vector::size) and cannot alter its elements.

### Parameters

none

### Return value

none

### Example

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 | // vector::shrink\_to\_fit  #include <iostream>  #include <vector>  int main ()  {  std::vector<int> myvector (100);  std::cout << "1. capacity of myvector: " << myvector.capacity() << '\n';  myvector.resize(10);  std::cout << "2. capacity of myvector: " << myvector.capacity() << '\n';  myvector.shrink\_to\_fit();  std::cout << "3. capacity of myvector: " << myvector.capacity() << '\n';  return 0;  } |

Possible output:

|  |
| --- |
| 1. capacity of myvector: 100  2. capacity of myvector: 100  3. capacity of myvector: 10 |

### Complexity

Up to linear in [size](http://www.cplusplus.com/vector::size).

### Iterator validity

If a reallocation happens, all iterators, pointers and references related to the container are invalidated.  
Otherwise, no changes.

### Data races

The container is modified.  
If a reallocation happens, all contained elements are modified.  
Otherwise, no contained elements are accessed.

### Exception safety

If the type of the elements is either copyable or no-throw moveable, there are no changes in the container in case of exception (strong guarantee).  
Otherwise, if an exception is thrown, the container is left with a valid state (basic guarantee).

public member function

<vector>

# std::[vector](http://www.cplusplus.com/reference/vector/vector/)::operator[]

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 [vector](http://www.cplusplus.com/vector) container.  
  
A similar member function, [vector::at](http://www.cplusplus.com/vector::at), has the same behavior as this operator function, except that [vector::at](http://www.cplusplus.com/vector::at) is bound-checked and signals if the requested position is *out of range* by throwing an [out\_of\_range](http://www.cplusplus.com/out_of_range) exception.   
  
Portable programs should never call this function with an argument *n* that is *out of range*, since this causes *undefined behavior*.

### Parameters

n

Position of an element in the container.  
Notice that the first element has a position of 0 (not 1).  
Member type size\_type is an unsigned integral type.

### Return value

The element at the specified position in the [vector](http://www.cplusplus.com/vector).  
  
If the [vector](http://www.cplusplus.com/vector) object is const-qualified, the function returns a const\_reference. Otherwise, it returns a reference.  
  
Member types reference and const\_reference are the reference types to the elements of the container (see [vector member types](http://www.cplusplus.com/vector#types)).

### Example

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 | // vector::operator[]  #include <iostream>  #include <vector>  int main ()  {  std::vector<int> myvector (10); // 10 zero-initialized elements  std::vector<int>::size\_type sz = myvector.size();  // assign some values:  for (unsigned i=0; i<sz; i++) myvector[i]=i;  // reverse vector using operator[]:  for (unsigned i=0; i<sz/2; i++)  {  int temp;  temp = myvector[sz-1-i];  myvector[sz-1-i]=myvector[i];  myvector[i]=temp;  }  std::cout << "myvector contains:";  for (unsigned i=0; i<sz; i++)  std::cout << ' ' << myvector[i];  std::cout << '\n';  return 0;  } |

Output:

|  |
| --- |
| myvector contains: 9 8 7 6 5 4 3 2 1 0 |

### Complexity

Constant.

### Iterator validity

No changes.

### Data races

The container is accessed (neither the const nor the non-const versions modify the container).  
The reference returned can be used to access or modify elements. Concurrently accessing or modifying different elements is safe.

### Exception safety

If the container [size](http://www.cplusplus.com/vector::size) is greater than *n*, the function never throws exceptions (no-throw guarantee).  
Otherwise, the behavior is undefined.

public member function

<vector>

# std::[vector](http://www.cplusplus.com/reference/vector/vector/)::at

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 [vector](http://www.cplusplus.com/vector).  
  
The function automatically checks whether *n* is within the bounds of valid elements in the [vector](http://www.cplusplus.com/vector), throwing an [out\_of\_range](http://www.cplusplus.com/out_of_range) exception if it is not (i.e., if *n* is greater or equal than its [size](http://www.cplusplus.com/vector::size)). This is in contrast with member [operator[]](http://www.cplusplus.com/vector::operator%5b%5d), that does not check against bounds.

### Parameters

n

Position of an element in the container.  
If this is greater than or equal to the [vector size](http://www.cplusplus.com/vector::size), an exception of type [out\_of\_range](http://www.cplusplus.com/out_of_range) is thrown.  
Notice that the first element has a position of 0 (not 1).  
Member type size\_type is an unsigned integral type.

### Return value

The element at the specified position in the container.  
  
If the [vector](http://www.cplusplus.com/vector) object is const-qualified, the function returns a const\_reference. Otherwise, it returns a reference.  
  
Member types reference and const\_reference are the reference types to the elements of the container (see [vector member types](http://www.cplusplus.com/vector#types)).

### Example

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 | // vector::at  #include <iostream>  #include <vector>  int main ()  {  std::vector<int> myvector (10); // 10 zero-initialized ints  // assign some values:  for (unsigned i=0; i<myvector.size(); i++)  myvector.at(i)=i;  std::cout << "myvector contains:";  for (unsigned i=0; i<myvector.size(); i++)  std::cout << ' ' << myvector.at(i);  std::cout << '\n';  return 0;  } |

Output:

|  |
| --- |
| myvector contains: 0 1 2 3 4 5 6 7 8 9 |

### Complexity

Constant.

### Iterator validity

No changes.

### Data races

The container is accessed (neither the const nor the non-const versions modify the container).  
The reference returned can be used to access or modify elements. Concurrently accessing or modifying different elements is safe.

### Exception safety

**Strong guarantee:** if an exception is thrown, there are no changes in the container.  
It throws [out\_of\_range](http://www.cplusplus.com/out_of_range) if *n* is out of bounds.

public member function

<vector>

# std::[vector](http://www.cplusplus.com/reference/vector/vector/)::front

reference front();

const\_reference front() const;

Access first element

Returns a reference to the first element in the [vector](http://www.cplusplus.com/vector).  
  
Unlike member [vector::begin](http://www.cplusplus.com/vector::begin), which returns an iterator to this same element, this function returns a direct reference.  
  
Calling this function on an [empty](http://www.cplusplus.com/vector::empty) container causes undefined behavior.

### Parameters

none

### Return value

A reference to the first element in the [vector](http://www.cplusplus.com/vector) container.  
  
If the [vector](http://www.cplusplus.com/vector) object is const-qualified, the function returns a const\_reference. Otherwise, it returns a reference.  
  
Member types reference and const\_reference are the reference types to the elements of the container (see [vector member types](http://www.cplusplus.com/vector#types)).

### Example

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 | // vector::front  #include <iostream>  #include <vector>  int main ()  {  std::vector<int> myvector;  myvector.push\_back(78);  myvector.push\_back(16);  // now front equals 78, and back 16  myvector.front() -= myvector.back();  std::cout << "myvector.front() is now " << myvector.front() << '\n';  return 0;  } |

Output:

|  |
| --- |
| myvector.front() is now 62 |

### Complexity

Constant.

### Iterator validity

No changes.

### Data races

The container is accessed (neither the const nor the non-const versions modify the container).  
The reference returned can be used to access or modify elements. Concurrently accessing or modifying different elements is safe.

### Exception safety

If the container is not [empty](http://www.cplusplus.com/vector::empty), the function never throws exceptions (no-throw guarantee).  
Otherwise, it causes *undefined behavior*.

public member function

<vector>

# std::[vector](http://www.cplusplus.com/reference/vector/vector/)::back

reference back();

const\_reference back() const;

Access last element

Returns a reference to the last element in the [vector](http://www.cplusplus.com/vector).  
  
Unlike member [vector::end](http://www.cplusplus.com/vector::end), which returns an iterator just past this element, this function returns a direct reference.  
  
Calling this function on an [empty](http://www.cplusplus.com/vector::empty) container causes undefined behavior.

### Parameters

none

### Return value

A reference to the last element in the [vector](http://www.cplusplus.com/vector).  
  
If the [vector](http://www.cplusplus.com/vector) object is const-qualified, the function returns a const\_reference. Otherwise, it returns a reference.  
  
Member types reference and const\_reference are the reference types to the elements of the [vector](http://www.cplusplus.com/vector) (see [member types](http://www.cplusplus.com/vector#types)).

### Example

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 | // vector::back  #include <iostream>  #include <vector>  int main ()  {  std::vector<int> myvector;  myvector.push\_back(10);  while (myvector.back() != 0)  {  myvector.push\_back ( myvector.back() -1 );  }  std::cout << "myvector contains:";  for (unsigned i=0; i<myvector.size() ; i++)  std::cout << ' ' << myvector[i];  std::cout << '\n';  return 0;  } |

Output:

|  |
| --- |
| myvector contains: 10 9 8 7 6 5 4 3 2 1 0 |

### Complexity

Constant.

### Iterator validity

No changes.

### Data races

The container is accessed (neither the const nor the non-const versions modify the container).  
The reference returned can be used to access or modify elements. Concurrently accessing or modifying different elements is safe.

### Exception safety

If the container is not [empty](http://www.cplusplus.com/vector::empty), the function never throws exceptions (no-throw guarantee).  
Otherwise, it causes *undefined behavior*.

public member function

<vector>

# std::[vector](http://www.cplusplus.com/reference/vector/vector/)::data

value\_type\* data() noexcept;

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

Access data

Returns a direct pointer to the memory array used internally by the [vector](http://www.cplusplus.com/vector) to store its owned elements.  
  
Because elements in the [vector](http://www.cplusplus.com/vector) are guaranteed to be stored in contiguous storage locations in the same order as represented by the [vector](http://www.cplusplus.com/vector), the pointer retrieved can be offset to access any element in the array.

### Parameters

none

### Return value

A pointer to the first element in the array used internally by the [vector](http://www.cplusplus.com/vector).  
  
If the [vector](http://www.cplusplus.com/vector) object is const-qualified, the function returns a pointer to const value\_type. Otherwise, it returns a pointer to value\_type.  
  
Member type value\_type is the type of the elements in the container, defined in [vector](http://www.cplusplus.com/vector) as an alias of the first class template parameter (T).

### Example

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 | // vector::data  #include <iostream>  #include <vector>  int main ()  {  std::vector<int> myvector (5);  int\* p = myvector.data();  \*p = 10;  ++p;  \*p = 20;  p[2] = 100;  std::cout << "myvector contains:";  for (unsigned i=0; i<myvector.size(); ++i)  std::cout << ' ' << myvector[i];  std::cout << '\n';  return 0;  } |

Output:

|  |
| --- |
| myvector contains: 10 20 0 100 0 |

### Complexity

Constant.

### Iterator validity

No changes.

### Data races

The container is accessed (neither the const nor the non-const versions modify the container).  
No contained elements are directly accessed by the call, but the pointer returned can be used to access or modify elements. Concurrently accessing or modifying different elements is safe.

### Exception safety

**No-throw guarantee:** this member function never throws exceptions.

public member function

<vector>

# std::[vector](http://www.cplusplus.com/reference/vector/vector/)::assign

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

|  |  |
| --- | --- |
| **range (1)** | template <class InputIterator>  void assign (InputIterator first, InputIterator last); |
| **fill (2)** | void assign (size\_type n, const value\_type& val); |

Assign vector content

Assigns new contents to the [vector](http://www.cplusplus.com/vector), replacing its current contents, and modifying its [size](http://www.cplusplus.com/vector::size) accordingly.

* [C++98](javascript:switch2.select(1))
* [C++11](javascript:switch2.select(2))

In the *range version* (1), the new contents are elements constructed from each of the elements in the range between *first* and *last*, in the same order.  
  
In the *fill version* (2), the new contents are *n* elements, each initialized to a copy of *val*.  
  
If a reallocation happens,the storage needed is allocated using the [internal allocator](http://www.cplusplus.com/vector::get_allocator).

Any elements held in the container before the call are *destroyed* and replaced by newly constructed elements (no assignments of elements take place).  
  
This causes an automatic reallocation of the allocated storage space if -and only if- the new vector [size](http://www.cplusplus.com/vector::size) surpasses the current vector [capacity](http://www.cplusplus.com/vector::capacity).

### Parameters

first, last

Input iterators to the initial and final positions in a sequence. The range used is [first,last), which includes all the elements between *first* and *last*, including the element pointed by *first* but not the element pointed by *last*.  
The function template argument InputIterator shall be an [input iterator](http://www.cplusplus.com/InputIterator) type that points to elements of a type from which value\_type objects can be constructed.

n

New size for the container.  
Member type size\_type is an unsigned integral type.

val

Value to fill the container with. Each of the *n* elements in the container will be initialized to a copy of this value.  
Member type value\_type is the type of the elements in the container, defined in [vector](http://www.cplusplus.com/vector) as an alias of its first template parameter (T).

il

An [initializer\_list](http://www.cplusplus.com/initializer_list) object. The compiler will automatically construct such objects from *initializer list* declarators.  
Member type value\_type is the type of the elements in the container, defined in [vector](http://www.cplusplus.com/vector) as an alias of its first template parameter (T).

### Return value

none

### Example

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 | // vector assign  #include <iostream>  #include <vector>  int main ()  {  std::vector<int> first;  std::vector<int> second;  std::vector<int> third;  first.assign (7,100); // 7 ints with a value of 100  std::vector<int>::iterator it;  it=first.begin()+1;  second.assign (it,first.end()-1); // the 5 central values of first  int myints[] = {1776,7,4};  third.assign (myints,myints+3); // assigning from array.  std::cout << "Size of first: " << int (first.size()) << '\n';  std::cout << "Size of second: " << int (second.size()) << '\n';  std::cout << "Size of third: " << int (third.size()) << '\n';  return 0;  } |

Output:

|  |
| --- |
| Size of first: 7  Size of second: 5  Size of third: 3 |

### Complexity

Linear on initial and final [sizes](http://www.cplusplus.com/vector::size) (destructions, constructions).  
Additionally, in the *range version (1)*, if InputIterator is not at least of a [forward iterator](http://www.cplusplus.com/ForwardIterator) category (i.e., it is just an [input iterator](http://www.cplusplus.com/InputIterator)) the new capacity cannot be determined beforehand and the operation incurs in additional logarithmic complexity in the new [size](http://www.cplusplus.com/vector::size) (reallocations while growing).

### Iterator validity

All iterators, pointers and references related to this container are invalidated.

### Data races

All copied elements are accessed.  
The container is modified.  
All contained elements are modified.

### Exception safety

**Basic guarantee:** if an exception is thrown, the container is in a valid state.  
If [allocator\_traits::construct](http://www.cplusplus.com/allocator_traits::construct) is not supported with the appropriate arguments for the element constructions, or if the range specified by [first,last) is not valid, it causes *undefined behavior*.

public member function

<vector>

# std::[vector](http://www.cplusplus.com/reference/vector/vector/)::push\_back

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

void push\_back (const value\_type& val);

Add element at the end

Adds a new element at the end of the [vector](http://www.cplusplus.com/vector), after its current last element. The content of *val* is copied (or moved) to the new element.  
  
This effectively increases the container [size](http://www.cplusplus.com/vector::size) by one, which causes an automatic reallocation of the allocated storage space if -and only if- the new vector [size](http://www.cplusplus.com/vector::size) surpasses the current vector [capacity](http://www.cplusplus.com/vector::capacity).

### Parameters

val

Value to be copied (or moved) to the new element.  
Member type value\_type is the type of the elements in the container, defined in [vector](http://www.cplusplus.com/vector) as an alias of its first template parameter (T).

### Return value

none  
  
If a reallocation happens, the storage is allocated using the container's [allocator](http://www.cplusplus.com/vector::get_allocator), which may throw exceptions on failure (for the default [allocator](http://www.cplusplus.com/allocator), bad\_alloc is thrown if the allocation request does not succeed).

### Example

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 | // vector::push\_back  #include <iostream>  #include <vector>  int main ()  {  std::vector<int> myvector;  int myint;  std::cout << "Please enter some integers (enter 0 to end):\n";  do {  std::cin >> myint;  myvector.push\_back (myint);  } while (myint);  std::cout << "myvector stores " << int(myvector.size()) << " numbers.\n";  return 0;  } |

The example uses push\_back to add a new element to the vector each time a new integer is read.

### Complexity

Constant (amortized time, reallocation may happen).  
  
If a reallocation happens, the reallocation is itself up to linear in the entire [size](http://www.cplusplus.com/vector::size).

### Iterator validity

If a reallocation happens, all iterators, pointers and references related to the container are invalidated.  
Otherwise, only the [end iterator](http://www.cplusplus.com/vector::end) is invalidated, and all iterators, pointers and references to elements are guaranteed to keep referring to the same elements they were referring to before the call.

### Data races

The container is modified.  
If a reallocation happens, all contained elements are modified.  
Otherwise, no existing element is accessed, and concurrently accessing or modifying them is safe.

### Exception safety

If no reallocations happen, there are no changes in the container in case of exception (strong guarantee).  
If a reallocation happens, the strong guarantee is also given if the type of the elements is either *copyable* or *no-throw moveable*.  
Otherwise, the container is guaranteed to end in a valid state (basic guarantee).  
If [allocator\_traits::construct](http://www.cplusplus.com/allocator_traits::construct) is not supported with *val* as argument, it causes *undefined behavior*.

public member function

<vector>

# std::[vector](http://www.cplusplus.com/reference/vector/vector/)::pop\_back

void pop\_back();

Delete last element

Removes the last element in the [vector](http://www.cplusplus.com/vector), effectively reducing the container [size](http://www.cplusplus.com/vector::size) by one.  
  
This destroys the removed element.

### Parameters

none

### Return value

none

### Example

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 | // vector::pop\_back  #include <iostream>  #include <vector>  int main ()  {  std::vector<int> myvector;  int sum (0);  myvector.push\_back (100);  myvector.push\_back (200);  myvector.push\_back (300);  while (!myvector.empty())  {  sum+=myvector.back();  myvector.pop\_back();  }  std::cout << "The elements of myvector add up to " << sum << '\n';  return 0;  } |

In this example, the elements are popped out of the [vector](http://www.cplusplus.com/vector) after they are added up in the sum. Output:

|  |
| --- |
| The elements of myvector add up to 600 |

### Complexity

Constant.

### Iterator validity

The [end iterator](http://www.cplusplus.com/vector::end) and any iterator, pointer and reference referring to the removed element are invalidated.  
Iterators, pointers and references referring to other elements that have not been removed are guaranteed to keep referring to the same elements they were referring to before the call.

### Data races

The container is modified.  
The last element is modified. Concurrently accessing or modifying other elements is safe, although iterating ranges that include the removed element is not.

### Exception safety

If the container is not [empty](http://www.cplusplus.com/vector::empty), the function never throws exceptions (no-throw guarantee).  
Otherwise, it causes *undefined behavior*.

public member function

<vector>

# std::[vector](http://www.cplusplus.com/reference/vector/vector/)::insert

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

|  |  |
| --- | --- |
| **single element (1)** | iterator insert (iterator position, const value\_type& val); |
| **fill (2)** | void insert (iterator position, size\_type n, const value\_type& val); |
| **range (3)** | template <class InputIterator>  void insert (iterator position, InputIterator first, InputIterator last); |

Insert elements

The [vector](http://www.cplusplus.com/vector) is extended by inserting new elements before the element at the specified *position*, effectively increasing the container [size](http://www.cplusplus.com/vector::size) by the number of elements inserted.  
  
This causes an automatic reallocation of the allocated storage space if -and only if- the new vector [size](http://www.cplusplus.com/vector::size) surpasses the current vector [capacity](http://www.cplusplus.com/vector::capacity).  
  
Because vectors use an array as their underlying storage, inserting elements in positions other than the [vector end](http://www.cplusplus.com/vector::end) causes the container to relocate all the elements that were after *position* to their new positions. This is generally an inefficient operation compared to the one performed for the same operation by other kinds of sequence containers (such as [list](http://www.cplusplus.com/list) or [forward\_list](http://www.cplusplus.com/forward_list)).  
  
The parameters determine how many elements are inserted and to which values they are initialized:

### Parameters

position

Position in the [vector](http://www.cplusplus.com/vector) where the new elements are inserted.  
iterator is a member type, defined as a [random access iterator](http://www.cplusplus.com/RandomAccessIterator) type that points to elements.

val

Value to be copied (or moved) to the inserted elements.  
Member type value\_type is the type of the elements in the container, defined in [deque](http://www.cplusplus.com/deque) as an alias of its first template parameter (T).

n

Number of elements to insert. Each element is initialized to a copy of *val*.  
Member type size\_type is an unsigned integral type.

first, last

Iterators specifying a range of elements. Copies of the elements in the range [first,last) are inserted at *position* (in the same order).  
Notice that the range includes all the elements between *first* and *last*, including the element pointed by *first* but not the one pointed by *last*.  
The function template argument InputIterator shall be an [input iterator](http://www.cplusplus.com/InputIterator) type that points to elements of a type from which value\_type objects can be constructed.

il

An [initializer\_list](http://www.cplusplus.com/initializer_list) object. Copies of these elements are inserted at *position* (in the same order).  
These objects are automatically constructed from *initializer list* declarators.  
Member type value\_type is the type of the elements in the container, defined in [vector](http://www.cplusplus.com/vector) as an alias of its first template parameter (T).

### Return value

An iterator that points to the first of the newly inserted elements.  
  
Member type iterator is a [random access iterator](http://www.cplusplus.com/RandomAccessIterator) type that points to elements.  
  
If reallocations happen, the storage is allocated using the container's [allocator](http://www.cplusplus.com/vector::get_allocator), which may throw exceptions on failure (for the default [allocator](http://www.cplusplus.com/allocator), bad\_alloc is thrown if the allocation request does not succeed).

### Example

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 | // inserting into a vector  #include <iostream>  #include <vector>  int main ()  {  std::vector<int> myvector (3,100);  std::vector<int>::iterator it;  it = myvector.begin();  it = myvector.insert ( it , 200 );  myvector.insert (it,2,300);  // "it" no longer valid, get a new one:  it = myvector.begin();  std::vector<int> anothervector (2,400);  myvector.insert (it+2,anothervector.begin(),anothervector.end());  int myarray [] = { 501,502,503 };  myvector.insert (myvector.begin(), myarray, myarray+3);  std::cout << "myvector contains:";  for (it=myvector.begin(); it<myvector.end(); it++)  std::cout << ' ' << \*it;  std::cout << '\n';  return 0;  } |

Output:

|  |
| --- |
| myvector contains: 501 502 503 300 300 400 400 200 100 100 100 |

### Complexity

Linear on the number of elements inserted (copy/move construction) plus the number of elements after *position* (moving).  
  
Additionally, if InputIterator in the *range insert (3)* is not at least of a [forward iterator](http://www.cplusplus.com/ForwardIterator) category (i.e., just an [input iterator](http://www.cplusplus.com/InputIterator)) the new capacity cannot be determined beforehand and the insertion incurs in additional logarithmic complexity in size (reallocations).  
  
If a reallocation happens, the reallocation is itself up to linear in the entire [size](http://www.cplusplus.com/vector::size) at the moment of the reallocation.

### Iterator validity

If a reallocation happens, all iterators, pointers and references related to the container are invalidated.  
Otherwise, only those pointing to *position* and beyond are invalidated, with all iterators, pointers and references to elements before *position* guaranteed to keep referring to the same elements they were referring to before the call.

### Data races

All copied elements are accessed.  
The container is modified.  
If a reallocation happens, all contained elements are modified.  
Otherwise, none of the elements before *position* is accessed, and concurrently accessing or modifying them is safe (although see *iterator validity* above).

### Exception safety

If the operation inserts a single element at the [end](http://www.cplusplus.com/vector::end), and no reallocations happen, there are no changes in the container in case of exception (strong guarantee). In case of reallocations, the strong guarantee is also given in this case if the type of the elements is either *copyable* or *no-throw moveable*.  
Otherwise, the container is guaranteed to end in a valid state (basic guarantee).  
If [allocator\_traits::construct](http://www.cplusplus.com/allocator_traits::construct) is not supported with the appropriate arguments for the element constructions, or if an invalid *position* or range is specified, it causes *undefined behavior*.

public member function

<vector>

# std::[vector](http://www.cplusplus.com/reference/vector/vector/)::erase

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

iterator erase (iterator position);

iterator erase (iterator first, iterator last);

Erase elements

Removes from the [vector](http://www.cplusplus.com/vector) either a single element (*position*) or a range of elements ([first,last)).  
  
This effectively reduces the container [size](http://www.cplusplus.com/vector::size) by the number of elements removed, which are destroyed.  
  
Because vectors use an array as their underlying storage, erasing elements in positions other than the [vector end](http://www.cplusplus.com/vector::end) causes the container to relocate all the elements after the segment erased to their new positions. This is generally an inefficient operation compared to the one performed for the same operation by other kinds of sequence containers (such as [list](http://www.cplusplus.com/list) or [forward\_list](http://www.cplusplus.com/forward_list)).

### Parameters

position

Iterator pointing to a single element to be removed from the [vector](http://www.cplusplus.com/vector).  
Member types iterator and const\_iterator are [random access iterator](http://www.cplusplus.com/RandomAccessIterator) types that point to elements.

first, last

Iterators specifying a range within the [vector](http://www.cplusplus.com/vector)] to be removed: [first,last). i.e., the range includes all the elements between *first* and *last*, including the element pointed by *first* but not the one pointed by *last*.  
Member types iterator and const\_iterator are [random access iterator](http://www.cplusplus.com/RandomAccessIterator) types that point to elements.

### Return value

An iterator pointing to the new location of the element that followed the last element erased by the function call. This is the [container end](http://www.cplusplus.com/vector::end) if the operation erased the last element in the sequence.  
  
Member type iterator is a [random access iterator](http://www.cplusplus.com/RandomAccessIterator) type that points to elements.

### Example

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 | // erasing from vector  #include <iostream>  #include <vector>  int main ()  {  std::vector<int> myvector;  // set some values (from 1 to 10)  for (int i=1; i<=10; i++) myvector.push\_back(i);  // erase the 6th element  myvector.erase (myvector.begin()+5);  // erase the first 3 elements:  myvector.erase (myvector.begin(),myvector.begin()+3);  std::cout << "myvector contains:";  for (unsigned i=0; i<myvector.size(); ++i)  std::cout << ' ' << myvector[i];  std::cout << '\n';  return 0;  } |

Output:

|  |
| --- |
| myvector contains: 4 5 7 8 9 10 |

### Complexity

Linear on the number of elements erased (destructions) plus the number of elements after the last element deleted (moving).

### Iterator validity

Iterators, pointers and references pointing to *position* (or *first*) and beyond are invalidated, with all iterators, pointers and references to elements before *position* (or *first*) are guaranteed to keep referring to the same elements they were referring to before the call.

### Data races

The container is modified.  
None of the elements before *position* (or *first*) is accessed, and concurrently accessing or modifying them is safe.

### Exception safety

If the removed elements include the last element in the container, no exceptions are thrown (no-throw guarantee).  
Otherwise, the container is guaranteed to end in a valid state (basic guarantee).  
An invalid *position* or range causes *undefined behavior*.

public member function

<vector>

# std::[vector](http://www.cplusplus.com/reference/vector/vector/)::swap

void swap (vector& x);

Swap content

Exchanges the content of the container by the content of *x*, which is another [vector](http://www.cplusplus.com/vector) object of the same type. Sizes may differ.  
  
After the call to this member function, the elements in this container are those which were in *x* before the call, and the elements of *x* are those which were in this. All iterators, references and pointers remain valid for the swapped objects.  
  
Notice that a non-member function exists with the same name, [swap](http://www.cplusplus.com/vector:swap), overloading that algorithm with an optimization that behaves like this member function.

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

No specifics on [allocators](http://www.cplusplus.com/vector::get_allocator). [contradictory specifications]

The bool specialization of [vector](http://www.cplusplus.com/vector) provides an additional overload for this function (see [vector<bool>::swap](http://www.cplusplus.com/vector%3Cbool%3E::swap)).

### Parameters

x

Another [vector](http://www.cplusplus.com/vector) container of the same type (i.e., instantiated with the same template parameters, T and Alloc) whose content is swapped with that of this container.

### Return value

none

### Example

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 | // swap vectors  #include <iostream>  #include <vector>  int main ()  {  std::vector<int> foo (3,100); // three ints with a value of 100  std::vector<int> bar (5,200); // five ints with a value of 200  foo.swap(bar);  std::cout << "foo contains:";  for (unsigned i=0; i<foo.size(); i++)  std::cout << ' ' << foo[i];  std::cout << '\n';  std::cout << "bar contains:";  for (unsigned i=0; i<bar.size(); i++)  std::cout << ' ' << bar[i];  std::cout << '\n';  return 0;  } |

Output:

|  |
| --- |
| foo contains: 200 200 200 200 200  bar contains: 100 100 100 |

### Complexity

Constant.

### Iterator validity

All iterators, pointers and references referring to elements in both containers remain valid, and are now referring to the same elements they referred to before the call, but in the other container, where they now iterate.  
Note that the *end iterators* do not refer to elements and may be invalidated.

### Data races

Both the container and *x* are modified.  
No contained elements are accessed by the call (although see *iterator validity* above).

### Exception safety

If the allocators in both [vectors](http://www.cplusplus.com/vector) compare equal, or if their [allocator traits](http://www.cplusplus.com/allocator_traits) indicate that the allocators shall [propagate](http://www.cplusplus.com/allocator_traits#types), the function never throws exceptions (no-throw guarantee).  
Otherwise, it causes *undefined behavior*.

public member function

<vector>

# std::[vector](http://www.cplusplus.com/reference/vector/vector/)::clear

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

void clear();

Clear content

Removes all elements from the [vector](http://www.cplusplus.com/vector) (which are destroyed), leaving the container with a [size](http://www.cplusplus.com/vector::size) of 0.  
  
A reallocation is not guaranteed to happen, and the [vector capacity](http://www.cplusplus.com/vector::capacity) is not guaranteed to change due to calling this function. A typical alternative that forces a reallocation is to use [swap](http://www.cplusplus.com/vector::swap):

|  |  |
| --- | --- |
|  | vector<T>().swap(x); // clear x reallocating |

### Parameters

none

### Return value

none

### Example

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 | // clearing vectors  #include <iostream>  #include <vector>  int main ()  {  std::vector<int> myvector;  myvector.push\_back (100);  myvector.push\_back (200);  myvector.push\_back (300);  std::cout << "myvector contains:";  for (unsigned i=0; i<myvector.size(); i++)  std::cout << ' ' << myvector[i];  std::cout << '\n';  myvector.clear();  myvector.push\_back (1101);  myvector.push\_back (2202);  std::cout << "myvector contains:";  for (unsigned i=0; i<myvector.size(); i++)  std::cout << ' ' << myvector[i];  std::cout << '\n';  return 0;  } |

Output:

|  |
| --- |
| myvector contains: 100 200 300  myvector contains: 1101 2202 |

### Complexity

Linear in [size](http://www.cplusplus.com/vector::size) (destructions).  
This may be optimized to *constant complexity* for [*trivially-destructible types*](http://www.cplusplus.com/is_trivially_destructible) (such as scalar or PODs), where elements need not be destroyed.

### Iterator validity

All iterators, pointers and references related to this container are invalidated.

### Data races

The container is modified.  
All contained elements are modified.

### Exception safety

**No-throw guarantee:** this member function never throws exceptions.

public member function

<vector>

# std::[vector](http://www.cplusplus.com/reference/vector/vector/)::emplace

template <class... Args>

iterator emplace (const\_iterator position, Args&&... args);

Construct and insert element

The container is extended by inserting a new element at *position*. This new element is constructed in place using *args* as the arguments for its construction.  
  
This effectively increases the container [size](http://www.cplusplus.com/vector::size) by one.  
  
An automatic reallocation of the allocated storage space happens if -and only if- the new vector [size](http://www.cplusplus.com/vector::size) surpasses the current vector [capacity](http://www.cplusplus.com/vector::capacity).  
  
Because vectors use an array as their underlying storage, inserting elements in positions other than the [vector end](http://www.cplusplus.com/vector::end) causes the container to shift all the elements that were after *position* by one to their new positions. This is generally an inefficient operation compared to the one performed by other kinds of sequence containers (such as [list](http://www.cplusplus.com/list) or [forward\_list](http://www.cplusplus.com/forward_list)). See [emplace\_back](http://www.cplusplus.com/vector::emplace_back) for a member function that extends the container directly at the [end](http://www.cplusplus.com/vector::end).  
  
The element is constructed in-place by calling [allocator\_traits::construct](http://www.cplusplus.com/allocator_traits::construct) with *args* forwarded.  
  
A similar member function exists, [insert](http://www.cplusplus.com/vector::insert), which either copies or moves existing objects into the container.

### Parameters

position

Position in the container where the new element is inserted.  
Member type const\_iterator is a [random access iterator](http://www.cplusplus.com/RandomAccessIterator) type that points to a const element.

args

Arguments forwarded to construct the new element.

### Return value

An iterator that points to the newly emplaced element.  
  
Member type iterator is a [random access iterator](http://www.cplusplus.com/RandomAccessIterator) type that points to an element.  
  
If a reallocation happens, the storage is allocated using the container's [allocator](http://www.cplusplus.com/vector::get_allocator), which may throw exceptions on failure (for the default [allocator](http://www.cplusplus.com/allocator), bad\_alloc is thrown if the allocation request does not succeed).

### Example

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 | // vector::emplace  #include <iostream>  #include <vector>  int main ()  {  std::vector<int> myvector = {10,20,30};  auto it = myvector.emplace ( myvector.begin()+1, 100 );  myvector.emplace ( it, 200 );  myvector.emplace ( myvector.end(), 300 );  std::cout << "myvector contains:";  for (auto& x: myvector)  std::cout << ' ' << x;  std::cout << '\n';  return 0;  } |

Output:

|  |
| --- |
| myvector contains: 10 200 100 20 30 300 |

### Complexity

Linear on the number of elements after *position* (moving).  
  
If a reallocation happens, the reallocation is itself up to linear in the entire [size](http://www.cplusplus.com/vector::size).

### Iterator validity

If a reallocation happens, all iterators, pointers and references related to this container are invalidated.  
Otherwise, only those pointing to *position* and beyond are invalidated, with all iterators, pointers and references to elements before *position* guaranteed to keep referring to the same elements they were referring to before the call.

### Data races

The container is modified.  
If a reallocation happens, all contained elements are modified.  
Otherwise, none of the elements before *position* is accessed, and concurrently accessing or modifying them is safe.

### Exception safety

If *position* is [end](http://www.cplusplus.com/vector::end), and no reallocations happen, there are no changes in the container in case of exception (strong guarantee).  
If a reallocation happens, the strong guarantee is also given if the type of the elements is either *copyable* or *no-throw moveable*.  
Otherwise, the container is guaranteed to end in a valid state (basic guarantee).  
If [allocator\_traits::construct](http://www.cplusplus.com/allocator_traits::construct) is not supported with the appropriate arguments, or if *position* is not valid, it causes *undefined behavior*.

public member function

<vector>

# std::[vector](http://www.cplusplus.com/reference/vector/vector/)::emplace\_back

template <class... Args>

void emplace\_back (Args&&... args);

Construct and insert element at the end

Inserts a new element at the end of the [vector](http://www.cplusplus.com/vector), right after its current last element. This new element is constructed in place using *args* as the arguments for its constructor.  
  
This effectively increases the container [size](http://www.cplusplus.com/vector::size) by one, which causes an automatic reallocation of the allocated storage space if -and only if- the new vector [size](http://www.cplusplus.com/vector::size) surpasses the current vector [capacity](http://www.cplusplus.com/vector::capacity).  
  
The element is constructed in-place by calling [allocator\_traits::construct](http://www.cplusplus.com/allocator_traits::construct) with *args* forwarded.  
  
A similar member function exists, [push\_back](http://www.cplusplus.com/vector::push_back), which either copies or moves an existing object into the container.

### Parameters

args

Arguments forwarded to construct the new element.

### Return value

none.  
  
If a reallocation happens, the storage is allocated using the container's [allocator](http://www.cplusplus.com/vector::get_allocator), which may throw exceptions on failure (for the default [allocator](http://www.cplusplus.com/allocator), bad\_alloc is thrown if the allocation request does not succeed).

### Example

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 | // vector::emplace\_back  #include <iostream>  #include <vector>  int main ()  {  std::vector<int> myvector = {10,20,30};  myvector.emplace\_back (100);  myvector.emplace\_back (200);  std::cout << "myvector contains:";  for (auto& x: myvector)  std::cout << ' ' << x;  std::cout << '\n';  return 0;  } |

Output:

|  |
| --- |
| myvector contains: 10 20 30 100 200 |

### Complexity

Constant (amortized time, reallocation may happen).  
  
If a reallocation happens, the reallocation is itself up to linear in the entire [size](http://www.cplusplus.com/vector::size).

### Iterator validity

If a reallocation happens, all iterators, pointers and references related to this container are invalidated.  
Otherwise, only the [end](http://www.cplusplus.com/vector::end) iterator is invalidated, and all other iterators, pointers and references to elements are guaranteed to keep referring to the same elements they were referring to before the call.

### Data races

The container is modified.  
If a reallocation happens, all contained elements are modified.  
Otherwise, no existing element is accessed, and concurrently accessing or modifying them is safe (although see *iterator validity* above).

### Exception safety

If no reallocations happen, there are no changes in the container in case of exception (strong guarantee).  
If a reallocation happens, the strong guarantee is also given if the type of the elements is either *copyable* or *no-throw moveable*.  
Otherwise, the container is guaranteed to end in a valid state (basic guarantee).  
If [allocator\_traits::construct](http://www.cplusplus.com/allocator_traits::construct) is not supported with the appropriate arguments, it causes *undefined behavior*.

public member function

<vector>

# std::[vector](http://www.cplusplus.com/reference/vector/vector/)::get\_allocator

* [C++98](javascript:switch1.select(1))
* [C++11](javascript:switch1.select(2))

allocator\_type get\_allocator() const;

Get allocator

Returns a copy of the allocator object associated with the [vector](http://www.cplusplus.com/vector).

### Parameters

none

### Return Value

The allocator.  
  
Member type allocator\_type is the type of the allocator used by the container, defined in [vector](http://www.cplusplus.com/vector) as an alias of its second template parameter (Alloc).

### Example

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 | // vector::get\_allocator  #include <iostream>  #include <vector>  int main ()  {  std::vector<int> myvector;  int \* p;  unsigned int i;  // allocate an array with space for 5 elements using vector's allocator:  p = myvector.get\_allocator().allocate(5);  // construct values in-place on the array:  for (i=0; i<5; i++) myvector.get\_allocator().construct(&p[i],i);  std::cout << "The allocated array contains:";  for (i=0; i<5; i++) std::cout << ' ' << p[i];  std::cout << '\n';  // destroy and deallocate:  for (i=0; i<5; i++) myvector.get\_allocator().destroy(&p[i]);  myvector.get\_allocator().deallocate(p,5);  return 0;  } |

The example shows an elaborate way to allocate memory for an array of ints using the same allocator used by the vector. Output:

|  |
| --- |
| The allocated array contains: 0 1 2 3 4 |

### Complexity

Constant.

### Iterator validity

No changes.

### Data races

The container is accessed.  
No contained elements are accessed: concurrently accessing or modifying them is safe.

### Exception safety

**No-throw guarantee:** this member function never throws exceptions.  
Copying any instantiation of the [default allocator](http://www.cplusplus.com/allocator) is also guaranteed to never throw.

function template

<vector>

# std::relational operators (vector)

|  |  |
| --- | --- |
| **(1)** | template <class T, class Alloc>  bool operator== (const vector<T,Alloc>& lhs, const vector<T,Alloc>& rhs); |
| **(2)** | template <class T, class Alloc>  bool operator!= (const vector<T,Alloc>& lhs, const vector<T,Alloc>& rhs); |
| **(3)** | template <class T, class Alloc>  bool operator< (const vector<T,Alloc>& lhs, const vector<T,Alloc>& rhs); |
| **(4)** | template <class T, class Alloc>  bool operator<= (const vector<T,Alloc>& lhs, const vector<T,Alloc>& rhs); |
| **(5)** | template <class T, class Alloc>  bool operator> (const vector<T,Alloc>& lhs, const vector<T,Alloc>& rhs);  (6)template <class T, class Alloc>  bool operator>= (const vector<T,Alloc>& lhs, const vector<T,Alloc>& rhs); |

Relational operators for vector

Performs the appropriate comparison operation between the [vector](http://www.cplusplus.com/vector) containers *lhs* and *rhs*.  
  
Operations == and != are performed by first comparing [sizes](http://www.cplusplus.com/vector::size), and if they match, the elements are compared sequentially using algorithm [equal](http://www.cplusplus.com/equal), which stops at the first mismatch.  
  
Operations <, >, <= and >= behave as if using algorithm [lexicographical\_compare](http://www.cplusplus.com/lexicographical_compare), which compares the elements sequentially using operator< reflexively, stopping at the first mismatch.  
  
These operators are overloaded in header [<vector>](http://www.cplusplus.com/%3Cvector%3E).

### Parameters

lhs, rhs

[vector](http://www.cplusplus.com/vector) containers (to the left- and right-hand side of the operator, respectively), having both the same template parameters (T and Alloc).

### Return Value

true if the condition holds, and false otherwise.

### Example

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 | // vector comparisons  #include <iostream>  #include <vector>  int main ()  {  std::vector<int> foo (3,100); // three ints with a value of 100  std::vector<int> bar (2,200); // two ints with a value of 200  if (foo==bar) std::cout << "foo and bar are equal\n";  if (foo!=bar) std::cout << "foo and bar are not equal\n";  if (foo< bar) std::cout << "foo is less than bar\n";  if (foo> bar) std::cout << "foo is greater than bar\n";  if (foo<=bar) std::cout << "foo is less than or equal to bar\n";  if (foo>=bar) std::cout << "foo is greater than or equal to bar\n";  return 0;  } |

Output:

|  |
| --- |
| foo and bar are not equal  foo is less than bar  foo is less than or equal to bar |

### Complexity

Linear in both *lhs* and *rhs*'s sizes.

### Iterator validity

No changes.

### Data races

Both containers, *lhs* and *rhs*, are accessed.  
Up to all of their contained elements may be accessed.  
In any case, the function cannot modify its arguments (const-qualified).

### Exception safety

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

function template

<vector>

# std::swap (vector)

template <class T, class Alloc>

void swap (vector<T,Alloc>& x, vector<T,Alloc>& y);

Exchange contents of vectors

The contents of container *x* are exchanged with those of *y*. Both container objects must be of the same type (same template parameters), although sizes may differ.  
  
After the call to this member function, the elements in *x* are those which were in *y* before the call, and the elements of *y* are those which were in *x*. All iterators, references and pointers remain valid for the swapped objects.  
  
This is an overload of the generic algorithm [swap](http://www.cplusplus.com/swap) that improves its performance by mutually transferring ownership over their assets to the other container (i.e., the containers exchange references to their data, without actually performing any element copy or movement): It behaves as if x.[swap](http://www.cplusplus.com/vector::swap)(y) was called.

### Parameters

x,y

[vector](http://www.cplusplus.com/vector) containers of the same type (i.e., having both the same template parameters, T and Alloc).

### Return value

none

### Example

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 | // swap (vector overload)  #include <iostream>  #include <vector>  main ()  {  unsigned int i;  std::vector<int> foo (3,100); // three ints with a value of 100  std::vector<int> bar (5,200); // five ints with a value of 200  foo.swap(bar);  std::cout << "foo contains:";  for (std::vector<int>::iterator it = foo.begin(); it!=foo.end(); ++it)  std::cout << ' ' << \*it;  std::cout << '\n';  std::cout << "bar contains:";  for (std::vector<int>::iterator it = bar.begin(); it!=bar.end(); ++it)  std::cout << ' ' << \*it;  std::cout << '\n';  return 0;  } |

Output:

|  |
| --- |
| foo contains: 200 200 200 200 200  bar contains: 100 100 100 |

### Complexity

Constant.

### Iterator validity

All iterators, pointers and references referring to elements in both containers remain valid, and are now referring to the same elements they referred to before the call, but in the other container, where they now iterate.  
Note that the [end iterator](http://www.cplusplus.com/vector::end) does not refer to an element and may be invalidated.

### Data races

Both containers, *x* and *y*, are modified.

### Exception safety

If the allocators in both [vectors](http://www.cplusplus.com/vector) compare equal, or if their [allocator traits](http://www.cplusplus.com/allocator_traits) indicate that the allocators shall [propagate](http://www.cplusplus.com/allocator_traits#types), the function never throws exceptions (no-throw guarantee).  
Otherwise, it causes *undefined behavior*.

class template specialization

<vector>

# std::vector<bool>

template < class T, class Alloc = allocator<T> > class vector; // generic template

template <class Alloc> class vector<bool,Alloc>; // bool specialization

Vector of bool

This is a specialized version of [vector](http://www.cplusplus.com/vector), which is used for elements of type bool and optimizes for space.  
  
It behaves like the unspecialized version of [vector](http://www.cplusplus.com/vector), with the following changes:

* The storage is not necessarily an array of bool values, but the library implementation may optimize storage so that each value is stored in a single bit.
* Elements are not constructed using the [allocator](http://www.cplusplus.com/vector::get_allocator) object, but their value is directly set on the proper bit in the internal storage.
* Member function [flip](http://www.cplusplus.com/vector%3Cbool%3E::flip) and a new signature for member [swap](http://www.cplusplus.com/vector%3Cbool%3E::swap).
* A special member type, [reference](http://www.cplusplus.com/vector%3Cbool%3E::reference), a class that accesses individual bits in the container's internal storage with an interface that emulates a bool reference. Conversely, member type const\_reference is a plain bool.
* The pointer and iterator types used by the container are not necessarily neither pointers nor conforming iterators, although they shall simulate most of their expected behavior.

These changes provide a quirky interface to this specialization and favor memory optimization over processing (which may or may not suit your needs). In any case, it is not possible to instantiate the unspecialized template of [vector](http://www.cplusplus.com/vector) for bool directly. Workarounds to avoid this range from using a different type (char, unsigned char) or container (like [deque](http://www.cplusplus.com/deque)) to use wrapper types or further specialize for specific allocator types.  
  
[bitset](http://www.cplusplus.com/bitset) is a class that provides a similar functionality for fixed-size arrays of bits.

### Template parameters

Alloc

Type of the allocator object used to define the storage allocation model. By default, [allocator](http://www.cplusplus.com/allocator)<bool> is used, which defines the simplest memory allocation model and is value-independent.  
Aliased as member type vector<bool>::allocator\_type.

### Member types

|  |  |  |
| --- | --- | --- |
| **member type** | **definition** | **notes** |
| value\_type | The first template parameter (bool) |  |
| allocator\_type | The second template parameter (Alloc) | defaults to: [allocator](http://www.cplusplus.com/allocator)<bool> |
| [reference](http://www.cplusplus.com/vector%3Cbool%3E::reference) | A specific member class (see [reference](http://www.cplusplus.com/vector%3Cbool%3E::reference) below) |  |
| const\_reference | bool |  |
| pointer | a type that simulates pointer behavior | convertible to const\_pointer |
| const\_pointer | a type that simulates pointer to const behavior |  |
| iterator | a type that simulates [random access iterator](http://www.cplusplus.com/RandomAccessIterator) behavior | convertible to const\_iterator |
| const\_iterator | a type that simulates [random access iterator](http://www.cplusplus.com/RandomAccessIterator) to const behavior |  |
| reverse\_iterator | [reverse\_iterator](http://www.cplusplus.com/reverse_iterator)<iterator> |  |
| const\_reverse\_iterator | [reverse\_iterator](http://www.cplusplus.com/reverse_iterator)<const\_iterator> |  |
| difference\_type | a signed integral type | usually the same as [ptrdiff\_t](http://www.cplusplus.com/ptrdiff_t) |
| size\_type | an unsigned integral type | usually the same as [size\_t](http://www.cplusplus.com/size_t) |

### Member classes

[**vector<bool>::reference**](http://www.cplusplus.com/reference/vector/vector-bool/reference/)

Reference type (public member class )

### Member functions

The specialization has the same member functions as the unspecialized [vector](http://www.cplusplus.com/vector), except [data](http://www.cplusplus.com/vector::data), [emplace](http://www.cplusplus.com/vector::emplace), and [emplace\_back](http://www.cplusplus.com/vector::emplace_back), that are not present in this specialization.  
  
It adds the following:

[**flip**](http://www.cplusplus.com/reference/vector/vector-bool/flip/)

Flip bits (public member function )

[**swap**](http://www.cplusplus.com/reference/vector/vector-bool/swap/)

Swap containers or elements (public member function )

### Non-member class specializations

[**hash<vector<bool>>**](http://www.cplusplus.com/reference/vector/vector-bool/hash/)

Hash for vector (class template specialization )