Vector

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

# Vector

* Vectors are sequence containers representing arrays that can change in size.
* vectors use contiguous storage locations for their elements
* their elements can also be accessed using offsets on regular pointers to its elements, and just as efficiently as in 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
* reallocation 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
* Vectors usually occupy more space than static arrays,
* because more memory is allocated to handle future growth
* thus the container may have an actual capacity greater than the size
* compared to arrays, vectors consume more memory
* Extra memory can be returned to the system via a call to shrink\_to\_fit(). (since C++11)
* vectors are very efficient accessing its elements (just like arrays) and relatively efficient adding or removing elements from its end
* Compared to the other dynamic sequence containers (deques, lists and forward\_lists)
* For insertion and removal of elements at positions other than the end, they perform worse than the others
* have less consistent iterators and references than lists and forward\_lists
* **complexity** (efficiency) of common operations on vectors is as follows:
* Random access - constant O(1)
* Insertion or removal of elements at the end - amortized constant O(1)
* Insertion or removal of elements - linear in the distance to the end of the vector O(n)

# 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

# Member functions

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

Construct vector (public member function )

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

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 )

# Member functions

## constructor

Construct vector (public member function )

|  |  |
| --- | --- |
| default (1) | explicit vector (const allocator\_type& alloc = allocator\_type());  //(until C++14) |
|  |  |
| fill (2) | explicit vector (size\_type n, const value\_type& val = value\_type(), const allocator\_type& alloc = allocator\_type()); //(until C++11)  explicit vector (size\_type n); //(since C++11)  vector (size\_type n, const value\_type& val, const allocator\_type& alloc = allocator\_type()); //(since C++11) |
|  |  |
| range (3) | template <class InputIterator>  vector (InputIterator first, InputIterator last, const allocator\_type& alloc = allocator\_type()); |
|  |  |
| copy (4) | vector (const vector& x);  vector (const vector& x, const allocator\_type& alloc);//(since C++11) |
|  |  |
| move (5) | vector (vector&& x);//(since C++11)  vector (vector&& x, const allocator\_type& alloc); //(since C++11) |
|  |  |
| Initialize list (6) | vector (initializer\_list<value\_type> il, const allocator\_type& alloc = allocator\_type()); //(since C++11) |

Constructs a vector, initializing its contents depending on the constructor version used

1. Constructs an empty container, with no elements.
2. Constructs a container with n elements. Each element is a copy of val (if provided).
3. Constructs a container with as many elements as the range [first,last),

with each element emplace-constructed from its corresponding element in that range, in the same order.

1. Constructs a container with a copy of each of the elements in x, in the same order.
2. Constructs a container that acquires the elements of x.

If alloc is specified and is different from x's allocator, the elements are moved.

Otherwise, no elements are constructed (their ownership is directly transferred).

x is left in an unspecified but valid state.

move constructors modify x.

1. Constructs a container with a copy of each of the elements in il, in the same order.

**Complexity**:

* Constant for the
  + default constructor (1), and
  + the move constructors (5) (unless alloc is different from x's allocator).
* For all other cases, linear in the resulting container size.

**Return**: (nothing)

**Data races**:

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

**Exception safety**:

* no effects in case an exception is thrown (Strong guarantee)
* it causes undefined behavior:
  + If 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

## destructor

Vector destructor (public member function )

|  |
| --- |
| ~vector(); |

* Vector destructor
* Destroys the container object.
* This calls allocator\_traits::destroy on each of the contained elements, and

deallocates all the storage capacity allocated by the vector using its allocator.

if the elements are pointers, the pointed-to objects are not destroyed.

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

**Return**:

**Data races**: The container and all its elements are modified.

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

## operator=

Assign content (public member function )

|  |  |
| --- | --- |
| copy (1) | vector& operator= (const vector& x); |
|  |  |
| move (2) | vector& operator= (vector&& x); //(since C++11)  vector& operator= (vector&& x) noexcept; //(since C++17) |
|  |  |
| initializer list (3) | vector& operator= (initializer\_list<value\_type> il); //(since C++11) |

* Assign content
* Assigns new contents to the container, replacing its current contents, and modifying its size accordingly.
* copy assignment (1) copies all the elements from x into the container (with x preserving its contents).
* move assignment (2) moves the elements of x into the container (x is left in an unspecified but valid state).
* initializer list assignment (3) copies the elements of il into the container.

**Complexity**: Linear in the size of \*this and other.

**Return**: \*this

**Data races**:

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

**Exception safety**:

* if an exception is thrown, the container is in a valid state (Basic guarantee)
* If allocator\_traits::construct is not supported with the appropriate arguments for the element constructions, or
* if value\_type is not copy assignable (or move assignable for (2)), it causes undefined behavior.
* no exception in case move(2) //(since C++17)

#include <iostream>

#include <vector>

using namespace std;

int main(void)

{

// constructors used in the same order as described above:

vector<int> first; // empty vector of ints

vector<int> second (4,100); // four ints with value 100

vector<int> third (second.begin(),second.end()); // iterating through second

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};

vector<int> fifth (myints, myints + sizeof(myints) / sizeof(int) );

cout << "The contents of fifth are:";

for (vector<int>::iterator it = fifth.begin(); it != fifth.end(); ++it)

cout << ' ' << \*it;

cout << "\n\n";

/\* operator= \*/

vector<int> foo (3,0);

vector<int> bar (5,0);

bar = foo;

foo = vector<int>();

cout << "Size of foo: " << int(foo.size()) << '\n';

cout << "Size of bar: " << int(bar.size()) << '\n';

return 0;

}

Output:

The contents of fifth are: 16 2 77 29

Size of foo: 0

Size of bar: 3

# Iterators:

## begin

Return iterator to beginning (public member function )

|  |
| --- |
| iterator begin(); //(until C++11) |
| const\_iterator begin() const; //(until C++11) |
|  |
| iterator begin() noexcept; //(since C++11) |
| const\_iterator begin() const noexcept; //(since C++11) |

* Return iterator to beginning
* Returns an iterator pointing to the first element in the vector.
* unlike member vector::front, which returns a reference to the first element,
* this function returns a random access iterator pointing to it.
* If the container is empty, the returned iterator will be equal to end().
* If the container is empty, the returned iterator value shall not be dereferenced.

**Complexity**: Constant

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

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

**Exception safety**:

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

## end

Return iterator to end (public member function )

|  |
| --- |
| iterator end();//(until C++11) |
| const\_iterator end() const; //(until C++11) |
|  |
| iterator end() noexcept; //(since C++11) |
| const\_iterator end() const noexcept;//(since C++11) |

* Return iterator to end
* Returns an iterator referring to the past-the-end element in the vector container.
* The past-the-end element is the theoretical element that would follow the last element in the vector.
* It does not point to any element, and thus shall not be dereferenced.
* This element acts as a placeholder; attempting to access it results in undefined behavior.
* 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 to specify a range including all the elements in the container.
* If the container is empty, this function returns the same as vector::begin.

**Complexity**: Constant

**Return**: An iterator to the element past the end of the sequence

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

**Exception safety**:

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

// vector::begin/end

#include <iostream>

#include <vector>

using namespace std;

int main ()

{

vector<int> myvector;

for (int i=1; i<=5; i++) myvector.push\_back(i);

cout << "myvector contains:";

for (vector<int>::iterator it = myvector.begin() ; it != myvector.end(); ++it)

cout << ' ' << \*it;

cout << '\n';

return 0;

}

Output:

myvector contains: 1 2 3 4 5

# Capacity:

## size

Return size (public member function )

|  |
| --- |
| size\_type size() const; //(until C++11) |
| size\_type size() const noexcept; //(since C++11) |

* Return size
* Returns the number of elements in the vector i.e. std::distance(begin(), end()).
* This is the number of actual objects held in the vector,
* which is not necessarily equal to its storage capacity.

**Complexity**: Constant

**Return**: number of elements in the container

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

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

## max\_size

Return maximum size (public member function )

|  |
| --- |
| size\_type max\_size() const; //(until C++11) |
| size\_type max\_size() const noexcept; //(since C++11) |

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

**Complexity**: Constant

**Return**: maximum number of elements a vector container can hold as content

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

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

## capacity

Return size of allocated storage capacity (public member function )

|  |
| --- |
| size\_type capacity() const; //(until C++11) |
| size\_type capacity() const noexcept; //(since C++11) |

* Return size of allocated storage capacity
* Returns the size of the storage space currently allocated for the vector, expressed in terms of elements.
* This capacity is not necessarily equal to the 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.
* 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 of a vector is given by member max\_size.
* The capacity of a vector can be explicitly altered by calling member vector::reserve.

**Complexity**: Constant

**Return**: Capacity of the currently allocated storage

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

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

## empty

Test whether vector is empty (public member function )

|  |
| --- |
| bool empty() const; //(until C++11) |
| bool empty() const noexcept; //(since C++11) |

* Checks if the container has no elements, i.e. whether begin() == end().
* Returns whether the vector is empty (i.e. whether its size is 0).
* This function does not modify the container in any way

**Complexity**: Constant

**Return**:

* true if the container is empty,
* false otherwise

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

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

#include <iostream>

#include <vector>

using namespace std;

int main ()

{

std::vector<int> myvector;

cout << "size: " << myvector.size() << "\n";

cout << "max\_size: " << myvector.max\_size() << "\n";

cout << "capacity: " << myvector.capacity() << "\n";

cout << "empty: " << myvector.empty() << "\n";

cout << "\n";

for (int i=0; i<33; i++) myvector.push\_back(i);

cout << "size: " << myvector.size() << "\n";

cout << "max\_size: " << myvector.max\_size() << "\n";

cout << "capacity: " << myvector.capacity() << "\n";

cout << "empty: " << myvector.empty() << "\n";

return 0;

}

Output:

size: 0

max\_size: 4611686018427387903

capacity: 0

empty: 1

size: 33

max\_size: 4611686018427387903

capacity: 64

empty: 0

## resize

Change size (public member function )

|  |
| --- |
| void resize (size\_type n, value\_type val = value\_type()); //(until C++11) |
| void resize (size\_type n); //(since C++11) |
| void resize (size\_type n, const value\_type& val); //(since C++11) |

* Change size
* Resizes the container so that it contains n elements.
* If **n < current container size**, the content is reduced to its first n elements, removing those beyond (and destroying them).
* If **n > current container 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 > current container 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.

**Complexity**:

* Linear in the difference between the current size and count
* Additional complexity possible due to reallocation if capacity is less than count

**Return**: (none)

**Data races**:

* 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 <= size** of the container, the function never throws exceptions (no-throw guarantee).
* If **n > size** 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)

## reserve

Request a change in capacity (public member function )

|  |
| --- |
| void reserve (size\_type n); |

* Request a change in capacity
* Increase the capacity of the vector to a value that's >= n
* If n > current vector capacity, the function causes the container to reallocate its storage increasing its capacity to n (or greater)
* In all other cases, the function call does not cause a reallocation and the vector capacity is not affected
* This function has no effect on the vector size and cannot alter its elements

**Complexity**: If a reallocation happens, linear in vector size at most.

**Return**: (none)

**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**:

**strong guarantee**: 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

**basic guarantee**: Otherwise, the container is guaranteed to end in a valid state

The function throws length\_error if n is greater than max\_size.

## shrink\_to\_fit

Shrink to fit (public member function )

|  |
| --- |
| void shrink\_to\_fit(); //(since C++11) |

* It is a non-binding request to reduce capacity() to size().
* It depends on the implementation if the request is fulfilled.
* The request is non-binding, and the container implementation is free to optimize

otherwise and leave the vector with a capacity greater than its size.

* This may cause a reallocation, but has no effect on the vector size and cannot alter its elements.

**Complexity**: At most, linear in container size

**Return**: (none)

**Data races**:

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

**Exception safety**:

**strong guarantee**: If the type of the elements is either copyable or no-throw moveable, there are no changes in the container in case of exception

**basic guarantee**: Otherwise, if an exception is thrown, the container is left with a valid state

# Element access

## operator[]

Access element (public member function )

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

* Access element
* Returns a reference to the element at position n in the vector container.
* Unlike std::map::operator[], this operator never inserts a new element into the container.
* No bounds checking is performed (vector::at does bound checking)
* Portable programs should never call this function with an argument n that is out of range, since this causes undefined behavior.

**Complexity**: Constant

**Return**:

* The element at the specified position in the vector
* If the vector object is const-qualified, the function returns a const\_reference
* Otherwise, it returns a reference.

**Data races**:

* The reference returned can be used to access or modify elements
* Concurrently accessing or modifying different elements is safe.

**Exception safety**:

* If container size > n, the function never throws exceptions (no-throw guarantee).
* Otherwise, the behavior is undefined.

## at

Access element (public member function )

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

* Access element
* Returns a reference to the element at position n in the vector, with bounds checking.
* If n is not within the range of the container (n >= size), an exception of type std::out\_of\_range is thrown.
* member operator[], does not perform bound checking

**Complexity**: Constant

**Return**:

* The element at the specified position in the container
* If the vector object is const-qualified, the function returns a const\_reference
* Otherwise, it returns a reference.

**Data races**:

* 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 if n is out of bounds.

## front

Access first element (public member function )

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

* Access first element
* Returns a reference to the first element in the vector.
* Calling front on an empty container is undefined.
* vector::begin, returns an iterator to the first element,
* vector::front, returns a direct reference to the first element,
* For a container c, the expression c.front() ≈ \*c.begin()

**Complexity**: Constant

**Return**:

* A reference to the first element in the vector container
* If the vector object is const-qualified, the function returns a const\_reference
* Otherwise, it returns a reference.

**Data races**:

* 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, the function never throws exceptions (no-throw guarantee).
* Otherwise, it causes undefined behavior.

## back

Access last element (public member function )

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

* Access last element
* Returns a reference to the last element in the vector.
* Calling this function on an empty container causes undefined behavior.
* vector::end, returns an iterator just past to the last element
* vector::back, returns a direct reference to the last element
* For a container c, the expression return

c.back()≈ { auto tmp = c.end(); --tmp; return \*tmp; }

**Complexity**: Constant

**Return**:

* A reference to the last element in the vector
* If the vector object is const-qualified, the function returns a const\_reference
* Otherwise, it returns a reference.

**Data races**:

* 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, the function never throws exceptions (no-throw guarantee).
* Otherwise, it causes undefined behavior.

## data

Access data (public member function )

|  |
| --- |
| 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 to store its owned elements.
* The pointer is such that range [data(); data() + size()) is always a valid range,
* even if the container is empty (data() is not dereferenceable in that case).
* Because elements in the vector are guaranteed to be stored in contiguous storage locations in the same order as represented by the vector,
* the pointer retrieved can be offset to access any element in the array.

**Complexity**: Constant

**Return**:

* A pointer to the first element in the array used internally by the vector
* If the vector object is const-qualified, the function returns a pointer to const value\_type.
* Otherwise, it returns a pointer to value\_type.

**Data races**:

* 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

#include <iostream>

#include <vector>

using namespace std;

int main () {

vector<int> myvector (5); // 5 zero-initialized elements

vector<int>::size\_type sz = myvector.size();

for (unsigned i=0; i<sz; i++) myvector[i]=i + 1;

cout << "myvector contains:";

for (unsigned i=0; i<sz; i++)

cout << ' ' << myvector[i];

cout << '\n';

cout << "myvector contains:";

for (unsigned i=0; i<sz; i++)

cout << ' ' << myvector.at(i);

cout << '\n';

/\*-----------------------------\*/

cout << "myvector.front() " << myvector.front() << '\n';

cout << "myvector.back() " << myvector.back() << '\n';

return 0;

}

Output:

myvector contains: 1 2 3 4 5

myvector contains: 1 2 3 4 5

myvector.front() 1

myvector.back() 5

// vector::data

#include <iostream>

#include <vector>

using namespace std;

int main () {

vector<int> myvector (5);

int\* p = myvector.data();

\*p = 10;

++p;

\*p = 20;

p[2] = 100;

cout << "myvector contains:";

for (unsigned i=0; i<myvector.size(); ++i)

cout << ' ' << myvector[i];

cout << '\n';

return 0;

}

Output:

myvector contains: 10 20 0 100 0

# Modifiers

## assign

Assign vector content (public member function )

|  |  |
| --- | --- |
| range (1) | template <class InputIterator>  void assign (InputIterator first, InputIterator last); |
|  |  |
| fill (2) | void assign (size\_type n, const value\_type& val); |
|  |  |
| initializer list (3) | void assign (initializer\_list<value\_type> il); //(since C++11) |

* Assign vector content
* Assigns new contents to the vector, replacing its current contents, and modifying its size accordingly.
* range (1), the new contents are elements constructed from each of the elements in the range between first and last, in the same order.
* fill (2), the new contents are n elements, each initialized to a copy of val.
* initializer list (3), the new contents are copies of the values passed as initializer list, in the same order.
* The internal allocator is used (through its traits) to allocate and deallocate storage if a reallocation happens.
* It is also used to destroy all existing elements, and to construct the new ones.
* 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 surpasses the current vector capacity.

**Complexity**:

1. Linear in count
2. Linear in distance between first and last
3. Linear in ilist.size()

in the range version (1), the operation incurs in additional logarithmic complexity in the new size (reallocations while growing)

**Return**: (none)

**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 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.

## push\_back

Add element at the end (public member function )

|  |  |
| --- | --- |
| 1 | void push\_back (const value\_type& val); |
| 2 | void push\_back (value\_type&& val); //(since C++11) |

* Add element at the end
* Adds a new element at the end of the vector, after its current last element.

1. The new element is initialized as a copy of value.
2. value is moved into the new element.

* This effectively increases the container size by one, which
* causes an automatic reallocation of the allocated storage space if new vector size > current vector capacity.

**Complexity**:

* Constant (amortized time, reallocation may happen).
* If a reallocation happens, the reallocation is itself up to linear in the entire size.

**Return**: (none)

**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**:

* **strong guarantee**: 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.
* **basic guarantee**: Otherwise, the container is guaranteed to end in a valid state
* If allocator\_traits::construct is not supported with val as argument, it causes undefined behavior.

## pop\_back

Delete last element (public member function )

|  |
| --- |
| void pop\_back(); |

* Delete last element
* Removes the last element in the vector, effectively reducing the container size by one.
* This destroys the removed element.
* Calling pop\_back on an empty container is undefined.
* No iterators or references except for back() and end() are invalidated.

**Complexity**: Constant

**Return**: (none)

**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**:

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

Otherwise, it causes undefined behavior.

Example - assign, push\_back, pop\_back

#include <iostream>

#include <vector>

using namespace std;

int main ()

{

vector<int> first;

vector<int> second;

vector<int> third;

first.assign (7,100); // 7 ints with a value of 100

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.

cout << "first vector: ";

for(int i = 0; i < first.size(); i++) cout << first.at(i) << " ";

cout << " size of first: " << int(first.size()) << '\n';

for(int i = 0; i < second.size(); i++) cout << second.at(i) << " ";

cout << " size of second: " << int(second.size()) << '\n';

for(int i = 0; i < third.size(); i++) cout << third.at(i) << " ";

cout << " size of third: " << int(third.size()) << '\n';

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

cout << '\n';

cout << "before push size: " << third.size() << " capacity: " << third.capacity() << '\n';

third.push\_back(13);

third.push\_back(17);

cout << "after push size: " << third.size() << " capacity: " << third.capacity() << '\n';

third.pop\_back();

third.pop\_back();

cout << "after pop size: " << third.size() << " capacity: " << third.capacity() << '\n';

return 0;

}

Output:

first vector: 100 100 100 100 100 100 100 size of first: 7

100 100 100 100 100 size of second: 5

1776 7 4 size of third: 3

before push size: 3 capacity: 3

after push size: 5 capacity: 6

after pop size: 3 capacity: 6

## insert

Insert elements (public member function )

|  |  |
| --- | --- |
| single element (1) | iterator insert (iterator position, const value\_type& val);  //(until C++11)  iterator insert (const\_iterator position, const value\_type& val); //(since C++11) |
|  |  |
| fill (2) | void insert (iterator position, size\_type n, const value\_type& val); //(until C++11)  iterator insert (const\_iterator position, size\_type n, const value\_type& val); //(since C++11) |
|  |  |
| range (3) | template <class InputIterator> //(until C++11)  void insert (iterator position, InputIterator first, InputIterator last);  template <class InputIterator> //(since C++11)  iterator insert (const\_iterator position, InputIterator first, InputIterator last); |
|  |  |
| move (4) | iterator insert (const\_iterator position, value\_type&& val);  //(since C++11) |
|  |  |
| initializer list (5) | iterator insert (const\_iterator position, initializer\_list<value\_type> il); //(since C++11) |

* Insert elements
* The vector is extended by inserting new elements before the element at the specified position,
* effectively increasing the container size by the number of elements inserted.
* inserts elements from range [first, last) before pos. (3)
* This causes an automatic reallocation if the new size() > old capacity()
* Because vectors use an array as their underlying storage,
* inserting elements in positions other than the 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 or forward\_list).

**Complexity**:

1. Constant plus linear in the distance between position and end of the container.
2. Linear in count plus linear in the distance between position and end of the container.
3. Linear in std::distance(first, last) plus linear in the distance between position and end of the container.
4. Constant plus linear in the distance between position and end of the container.
5. Linear in ilist.size() plus linear in the distance between position and end of the container.

**Return**:

1. Iterator pointing to the inserted value
2. Iterator pointing to the first element inserted, or position if count==0.
3. Iterator pointing to the first element inserted, or position if first==last.
4. Iterator pointing to the inserted value
5. Iterator pointing to the first element inserted, or position if ilist is empty.

**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 saf

**Exception safety**:

* If the operation inserts a single element at the 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 is not supported with the appropriate arguments for the element constructions, or if an invalid position or range is specified, it causes undefined behavior.

#include <iostream>

#include <vector>

using namespace std;

int main ()

{

vector<int> myvector (3,100);

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();

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);

cout << "myvector contains:";

for (it=myvector.begin(); it<myvector.end(); it++)

cout << ' ' << \*it;

cout << '\n';

return 0;

}

Output:

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

## erase

Erase elements (public member function )

|  |
| --- |
| iterator erase (iterator position); //(until C++11) |
| iterator erase (iterator first, iterator last); //(until C++11) |
|  |
| iterator erase (const\_iterator position); //(since C++11) |
| iterator erase (const\_iterator first, const\_iterator last); //(since C++11) |

* Erase elements
* Removes from the vector either
  + a single element (position) or
  + a range of elements ([first,last)).
* This effectively reduces the container size by the number of elements removed, which are destroyed.
* The iterator position must be valid and dereferenceable.
* Thus the end() iterator cannot be used as a value for pos. (bcoz it is valid, but is not dereferencable)
* The iterator first does not need to be dereferenceable if first==last: erasing an empty range is a no-op.
* Because vectors use an array as their underlying storage, erasing elements in positions other than the 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 or forward\_list).

**Complexity**:

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

**Return**:

* 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 if the operation erased the last element in the sequence.
* Member type iterator is a random access iterator type that points to elements.

**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.

## swap

Swap content (public member function )

|  |
| --- |
| void swap (vector& x); //(until C++17) |
| void swap (vector& x) noexcept; //(since C++17) |

* Swap content
* Exchanges the contents of the container with those of other container x
* Does not invoke any move, copy, or swap operations on individual elements.
* 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,
* overloading that algorithm with an optimization that behaves like this member function.
* No specifics on allocators. [contradictory specifications] //(until C++11)
* Whether the container allocators are also swapped is not defined,
* unless in the case the appropriate allocator traits indicate explicitly that they shall propagate. //(since C++11)

**Complexity**: Constant

**Return**: (none)

**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 compare equal, or if their allocator traits indicate that the allocators shall propagate,
* the function never throws exceptions (no-throw guarantee).
* Otherwise, it causes undefined behavior.

## clear

Clear content (public member function )

|  |
| --- |
| void clear(); //(until C++11) |
| void clear() noexcept; //(since C++11) |

* Clear content
* Removes all elements from the vector (which are destroyed), leaving the container with a size of 0.
* A reallocation is not guaranteed to happen, and the vector capacity is not guaranteed to change due to calling this function.
* A typical alternative that forces a reallocation is to use swap:
* vector<T>().swap(x); // clear x reallocating

**Complexity**:

* Linear in size (destructions)
* This may be optimized to constant complexity for trivially-destructible types (such as scalar or PODs), where elements need not be destroyed.

**Return**: (none)

**Data races**:

* The container is modified.
* All contained elements are modified.

**Exception safety**:

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

Example : erase, size and clear

#include <iostream>

#include <vector>

using namespace std;

int main ()

{

vector<int> myvector;

for (int i=1; i<=10; i++) myvector.push\_back(i);

// erase the 6th element

myvector.erase (myvector.begin()+5);

cout << "myvector contains:";

for (unsigned i=0; i<myvector.size(); ++i)

cout << ' ' << myvector[i];

cout << '\n';

// erase the first 3 elements:

myvector.erase (myvector.begin(),myvector.begin()+3);

cout << "myvector contains:";

for (unsigned i=0; i<myvector.size(); ++i)

cout << ' ' << myvector[i];

cout << '\n';

/\*\*\*\*\*\*\*\*\*\*\*/

vector<int> foo (3,100);

vector<int> bar (5,200);

foo.swap(bar);

cout << "foo contains:";

for (unsigned i=0; i<foo.size(); i++)

cout << ' ' << foo[i];

cout << '\n';

cout << "bar contains:";

for (unsigned i=0; i<bar.size(); i++)

cout << ' ' << bar[i];

cout << '\n';

/\*\*\*\*\*\*\*\*\*\*\*/

foo.clear();

cout << "foo contains:";

for (unsigned i=0; i<foo.size(); i++)

cout << ' ' << foo[i];

cout << '\n';

foo.push\_back(1101);

foo.push\_back(2202);

cout << "foo contains:";

for (unsigned i=0; i<foo.size(); i++)

cout << ' ' << foo[i];

cout << '\n';

return 0;

}

Output:

myvector contains: 1 2 3 4 5 7 8 9 10

myvector contains: 4 5 7 8 9 10

foo contains: 200 200 200 200 200

bar contains: 100 100 100

foo contains:

foo contains: 1101 2202

## emplace

Construct and insert element (public member function )

|  |
| --- |
| template <class... Args>  iterator emplace (const\_iterator position, Args&&... args); //(since C++11) |

* Construct and insert element into the container directly before position
* 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 by one.
* An automatic reallocation of the allocated storage space happens If the new size() > capacity()
* Because vectors use an array as their underlying storage,
* inserting elements in positions other than the 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 or forward\_list).
* (emplace\_back extends the container directly at the end.)
* The element is constructed in-place by calling allocator\_traits::construct with args forwarded.
* A similar member function exists, insert, which either copies or moves existing objects into the container.

**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.

**Return**:

An iterator that points to the newly emplaced element

Member type iterator is a random access iterator type that points to an element.

**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**:

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

Example

#include <iostream>

#include <vector>

using namespace std;

int main () {

vector<int> myvector = {10,20,30};

auto it = myvector.emplace ( myvector.begin()+1, 100 );

cout << "myvector contains:";

for (auto& x: myvector)

cout << ' ' << x;

cout << '\n';

myvector.emplace ( it, 200 );

cout << "myvector contains:";

for (auto& x: myvector)

cout << ' ' << x;

cout << '\n';

myvector.emplace ( myvector.end(), 300 );

cout << "myvector contains:";

for (auto& x: myvector)

cout << ' ' << x;

cout << '\n';

return 0;

}

Output:

myvector contains: 10 100 20 30

myvector contains: 10 200 100 20 30

myvector contains: 10 200 100 20 30 300

## emplace\_back

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

|  |
| --- |
| template <class... Args>  void emplace\_back (Args&&... args);//(since C++11) |

* Construct and insert element to the end of the container
* Inserts a new element at the end of the 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 by one,
* It causes an automatic reallocation of the allocated storage space if new vector size() > current vector capacity()
* The element is constructed in-place by calling allocator\_traits::construct with args forwarded.
* A similar member function exists, push\_back, which either copies or moves an existing object into the container.

**Complexity**:

Constant (amortized time, reallocation may happen).

If a reallocation happens, the reallocation is itself up to linear in the entire size.

**Return**: (none)

**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**:

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

Example:

#include <iostream>

#include <vector>

using namespace std;

int main () {

vector<int> myvector = {10,20,30};

cout << "myvector contains:";

for (auto& x: myvector)

cout << ' ' << x;

cout << '\n';

myvector.emplace\_back (100);

cout << "myvector contains:";

for (auto& x: myvector)

cout << ' ' << x;

cout << '\n';

myvector.emplace\_back (200);

cout << "myvector contains:";

for (auto& x: myvector)

cout << ' ' << x;

cout << '\n';

return 0;

}

Output:

myvector contains: 10 20 30

myvector contains: 10 20 30 100

myvector contains: 10 20 30 100 200

# Allocator:

## get\_allocator

Get allocator (public member function )

|  |
| --- |
| allocator\_type get\_allocator() const; //(until C++11) |
| allocator\_type get\_allocator() const noexcept; //(since C++11) |

* Get allocator
* Returns a copy of the allocator object associated with the vector

**Complexity**: Constant

**Return**: The allocator

**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 is also guaranteed to never throw.

#include <iostream>

#include <vector>

using namespace std;

int main ()

{

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);

cout << "The allocated array contains:";

for (i=0; i<5; i++) cout << ' ' << p[i];

cout << '\n';

// construct values in-place on the array:

for (i=0; i<5; i++) myvector.get\_allocator().construct(&p[i],i);

cout << "The allocated array contains:";

for (i=0; i<5; i++) cout << ' ' << p[i];

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;

}

Output:

The allocated array contains: 0 0 0 0 0

The allocated array contains: 0 1 2 3 4

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**END**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*