vector Class

Visual Studio 2015

The new home for Visual Studio documentation is Visual Studio 2017 Documentation on docs.microsoft.com.

The latest version of this topic can be found at vector Class.

The STL vector class is a template class of sequence containers that arrange elements of a given type in a linear arrangement and allow fast random access to any element. They should be the preferred container for a sequence when random-access performance is at a premium.

Syntax

```
template <class Type, class Allocator = allocator<Type>>
class vector
```

Parameters

Type
The element data type to be stored in the vector

`Allocator`
The type that represents the stored allocator object that encapsulates details about the vector's allocation and deallocation of memory. This argument is optional and the default value is **allocator**<*Type*>.

Remarks

Vectors allow constant time insertions and deletions at the end of the sequence. Inserting or deleting elements in the middle of a vector requires linear time. The performance of the deque Class container is superior with respect to insertions and deletions at the beginning and end of a sequence. The list Class container is superior with respect to insertions and deletions at any location within a sequence.

Vector reallocation occurs when a member function must increase the sequence contained in the vector object beyond its current storage capacity. Other insertions and erasures may alter various storage addresses within the sequence. In all such cases, iterators or references that point at altered portions of the sequence become invalid. If no reallocation happens, only iterators and references before the insertion/deletion point remain valid.

The vector<bool> Class is a full specialization of the template class vector for elements of type bool with an allocator for the underlying type used by the specialization.

The vector<bool> reference Class is a nested class whose objects are able to provide references to elements (single bits) within a vector<bool> object.

Members

Constructors

vector	Constructs a vector of a specific size or with elements of a specific value or with a specific `allocator` or as a copy of some other vector.

Typedefs

allocator_type	A type that represents the `allocator` class for the vector object.
const_iterator	A type that provides a random-access iterator that can read a `const` element in a vector.
const_pointer	A type that provides a pointer to a `const` element in a vector.
const_reference	A type that provides a reference to a `const` element stored in a vector for reading and performing `const` operations.
const_reverse_iterator	A type that provides a random-access iterator that can read any `const` element in the vector.
difference_type	A type that provides the difference between the addresses of two elements in a vector.
iterator	A type that provides a random-access iterator that can read or modify any element in a vector.
pointer	A type that provides a pointer to an element in a vector.
reference	A type that provides a reference to an element stored in a vector.
reverse_iterator	A type that provides a random-access iterator that can read or modify any element in a reversed vector.
size_type	A type that counts the number of elements in a vector.
value_type	A type that represents the data type stored in a vector.

Member Functions

assign	Erases a vector and copies the specified elements to the empty vector.
at	Returns a reference to the element at a specified location in the vector.
back	Returns a reference to the last element of the vector.
begin	Returns a random-access iterator to the first element in the vector.
capacity	Returns the number of elements that the vector could contain without allocating more storage.
cbegin	Returns a random-access const iterator to the first element in the vector.
cend	Returns a random-access const iterator that points just beyond the end of the vector.
crbegin	Returns a const iterator to the first element in a reversed vector.
crend	Returns a const iterator to the end of a reversed vector.
clear	Erases the elements of the vector.
data	Returns a pointer to the first element in the vector.
emplace	Inserts an element constructed in place into the vector at a specified position.
emplace_back	Adds an element constructed in place to the end of the vector.
empty	Tests if the vector container is empty.
end	Returns a random-access iterator that points to the end of the vector.
erase	Removes an element or a range of elements in a vector from specified positions.
front	Returns a reference to the first element in a vector.
get_allocator	Returns an object to the `allocator` class used by a vector.
insert	Inserts an element or a number of elements into the vector at a specified position.
max_size	Returns the maximum length of the vector.
pop_back	Deletes the element at the end of the vector.
push_back	Add an element to the end of the vector.
rbegin	Returns an iterator to the first element in a reversed vector.
rend	Returns an iterator to the end of a reversed vector.

reserve	Reserves a minimum length of storage for a vector object.
resize	Specifies a new size for a vector.
shrink_to_fit	Discards excess capacity.
size	Returns the number of elements in the vector.
swap	Exchanges the elements of two vectors.

Operators

operator[]	Returns a reference to the vector element at a specified position.
operator=	Replaces the elements of the vector with a copy of another vector.

Requirements

Header: <vector>

Namespace: std

vector::allocator_type

A type that represents the allocator class for the vector object.

```
typedef Allocator allocator_type;
```

Remarks

`allocator_type` is a synonym for the template parameter **Allocator.**

Example

See the example for get_allocator for an example that uses `allocator_type`.

vector::assign

Erases a vector and copies the specified elements to the empty vector.

```
void assign(
    size_type Count,
    const Type& Val);

void assign(
    initializer_list<Type> IList);

template <class InputIterator>
void assign(
    InputIterator First,
    InputIterator Last);
```

Parameters

`First`
Position of the first element in the range of elements to be copied.

`Last`
Position of the first element beyond the range of elements to be copied.

`Count`
The number of copies of an element being inserted into the vector.

`Val`
The value of the element being inserted into the vector.

`IList`
The initializer_list containing the elements to insert.

Remarks

After erasing any existing elements in a vector, assign either inserts a specified range of elements from the original vector into a vector or inserts copies of a new element of a specified value into a vector.

Example

```
/ vector_assign.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main()
{
    using namespace std;
    vector<int> v1, v2, v3;

    v1.push_back(10);
    v1.push_back(20);
    v1.push_back(30);
    v1.push_back(40);
```

```
    v1.push_back(50);

    cout << "v1 = ";
    for (auto& v : v1){
        cout << v << " ";
    }
    cout << endl;

    v2.assign(v1.begin(), v1.end());
    cout << "v2 = ";
    for (auto& v : v2){
        cout << v << " ";
    }
    cout << endl;

    v3.assign(7, 4);
    cout << "v3 = ";
    for (auto& v : v3){
        cout << v << " ";
    }
    cout << endl;

    v3.assign({ 5, 6, 7 });
    for (auto& v : v3){
        cout << v << " ";
    }
    cout << endl;
}
```

vector::at

Returns a reference to the element at a specified location in the vector.

```
reference at(size_type _Pos);

const_reference at(size_type _Pos) const;
```

Parameters

`_Pos`
The subscript or position number of the element to reference in the vector.

Return Value

A reference to the element subscripted in the argument. If `_Off` is greater than the size of the vector, **at** throws an exception.

Remarks

If the return value of **at** is assigned to a `const_reference`, the vector object cannot be modified. If the return value of **at** is assigned to a **reference**, the vector object can be modified.

Example

```
// vector_at.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;

   v1.push_back( 10 );
   v1.push_back( 20 );

   const int &i = v1.at( 0 );
   int &j = v1.at( 1 );
   cout << "The first element is " << i << endl;
   cout << "The second element is " << j << endl;
}
```

Output

```
The first element is 10
The second element is 20
```

vector::back

Returns a reference to the last element of the vector.

```
reference back();

const_reference back() const;
```

Return Value

The last element of the vector. If the vector is empty, the return value is undefined.

Remarks

If the return value of **back** is assigned to a `const_reference`, the vector object cannot be modified. If the return value of **back** is assigned to a **reference**, the vector object can be modified.

When compiled by using `_ITERATOR_DEBUG_LEVEL` defined as 1 or 2, a runtime error will occur if you attempt to access an element in an empty vector. See Checked Iterators for more information.

Example

```
// vector_back.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main() {
   using namespace std;
   vector <int> v1;

   v1.push_back( 10 );
   v1.push_back( 11 );

   int& i = v1.back( );
   const int& ii = v1.front( );

   cout << "The last integer of v1 is " << i << endl;
   i--;
   cout << "The next-to-last integer of v1 is "<< ii << endl;
}
```

vector::begin

Returns a random-access iterator to the first element in the vector.

```
const_iterator begin() const;


iterator begin();
```

Return Value

A random-access iterator addressing the first element in the `vector` or to the location succeeding an empty `vector`. You should always compare the value returned with vector::end to ensure it is valid.

Remarks

If the return value of `begin` is assigned to a vector::const_iterator, the `vector` object cannot be modified. If the return value of `begin` is assigned to an vector::iterator, the `vector` object can be modified.

Example

```
// vector_begin.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main()
{
    using namespace std;
    vector<int> c1;
    vector<int>::iterator c1_Iter;
    vector<int>::const_iterator c1_cIter;

    c1.push_back(1);
    c1.push_back(2);

    cout << "The vector c1 contains elements:";
    c1_Iter = c1.begin();
    for (; c1_Iter != c1.end(); c1_Iter++)
    {
        cout << " " << *c1_Iter;
    }
    cout << endl;

    cout << "The vector c1 now contains elements:";
    c1_Iter = c1.begin();
 *c1_Iter = 20;
    for (; c1_Iter != c1.end(); c1_Iter++)
    {
        cout << " " << *c1_Iter;
    }
    cout << endl;

    // The following line would be an error because iterator is const
    // *c1_cIter = 200;
}
```

Output

```
The vector c1 contains elements: 1 2
The vector c1 now contains elements: 20 2
```

vector::capacity

Returns the number of elements that the vector could contain without allocating more storage.

```
size_type capacity() const;
```

Return Value

The current length of storage allocated for the vector.

Remarks

The member function resize will be more efficient if sufficient memory is allocated to accommodate it. Use the member function reserve to specify the amount of memory allocated.

Example

```
// vector_capacity.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;

   v1.push_back( 1 );
   cout << "The length of storage allocated is "
        << v1.capacity( ) << "." << endl;

   v1.push_back( 2 );
   cout << "The length of storage allocated is now "
        << v1.capacity( ) << "." << endl;
}
```

Output

```
The length of storage allocated is 1.
The length of storage allocated is now 2.
```

vector::cbegin

Returns a const iterator that addresses the first element in the range.

```
const_iterator cbegin() const;
```

Return Value

A `const` random-access iterator that points at the first element of the range, or the location just beyond the end of an empty range (for an empty range, `cbegin() == cend()`).

Remarks

With the return value of `cbegin`, the elements in the range cannot be modified.

You can use this member function in place of the `begin()` member function to guarantee that the return value is `const_iterator`. Typically, it's used in conjunction with the auto type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- `const`) container of any kind that supports `begin()` and `cbegin()`.

C++

```
auto i1 = Container.begin();
// i1 is Container<T>::iterator
auto i2 = Container.cbegin();

// i2 is Container<T>::const_iterator
```

vector::cend

Returns a `const` iterator that addresses the location just beyond the last element in a range.

```
const_iterator cend() const;
```

Return Value

A `const` random-access iterator that points just beyond the end of the range.

Remarks

cend is used to test whether an iterator has passed the end of its range.

You can use this member function in place of the `end()` member function to guarantee that the return value is `const_iterator`. Typically, it's used in conjunction with the auto type deduction keyword, as shown in the following example. In the example, consider `Container` to be a modifiable (non- `const`) container of any kind that supports `end()` and `cend()`.

C++

```
auto i1 = Container.end();
```

```
// i1 is Container<T>::iterator
auto i2 = Container.cend();

// i2 is Container<T>::const_iterator
```

The value returned by cend should not be dereferenced.

vector::clear

Erases the elements of the vector.

```
void clear();
```

Example

```
// vector_clear.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;

   v1.push_back( 10 );
   v1.push_back( 20 );
   v1.push_back( 30 );

   cout << "The size of v1 is " << v1.size( ) << endl;
   v1.clear( );
   cout << "The size of v1 after clearing is " << v1.size( ) << endl;
}
```

Output

```
The size of v1 is 3
The size of v1 after clearing is 0
```

vector::const_iterator

A type that provides a random-access iterator that can read a **const** element in a vector.

```
typedef implementation-defined const_iterator;
```

Remarks

A type `const_iterator` cannot be used to modify the value of an element.

Example

See the example for back for an example that uses `const_iterator`.

vector::const_pointer

A type that provides a pointer to a **const** element in a vector.

```
typedef typename Allocator::const_pointer const_pointer;
```

Remarks

A type `const_pointer` cannot be used to modify the value of an element.

An iterator is more commonly used to access a vector element.

vector::const_reference

A type that provides a reference to a **const** element stored in a vector for reading and performing **const** operations.

```
typedef typename Allocator::const_reference const_reference;
```

Remarks

A type `const_reference` cannot be used to modify the value of an element.

Example

```
```

```
// vector_const_ref.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;

   v1.push_back( 10 );
   v1.push_back( 20 );

   const vector <int> v2 = v1;
   const int &i = v2.front( );
   const int &j = v2.back( );
   cout << "The first element is " << i << endl;
   cout << "The second element is " << j << endl;

   // The following line would cause an error as v2 is const
   // v2.push_back( 30 );
}
```

Output

```
The first element is 10
The second element is 20
```

vector::const_reverse_iterator

A type that provides a random-access iterator that can read any **const** element in the vector.

```
typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
```

Remarks

A type `const_reverse_iterator` cannot modify the value of an element and is used to iterate through the vector in reverse.

Example

See rbegin for an example of how to declare and use an iterator.

vector::crbegin

Returns a const iterator to the first element in a reversed vector.

```
const_reverse_iterator crbegin() const;
```

Return Value

A const reverse random-access iterator addressing the first element in a reversed vector or addressing what had been the last element in the unreversed `vector`.

Remarks

With the return value of `crbegin`, the `vector` object cannot be modified.

Example

```
// vector_crbegin.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;
   vector <int>::iterator v1_Iter;
   vector <int>::const_reverse_iterator v1_rIter;

   v1.push_back( 1 );
   v1.push_back( 2 );

   v1_Iter = v1.begin( );
   cout << "The first element of vector is "
        << *v1_Iter << "." << endl;

   v1_rIter = v1.crbegin( );
   cout << "The first element of the reversed vector is "
        << *v1_rIter << "." << endl;
}
```

Output

```
The first element of vector is 1.
The first element of the reversed vector is 2.
```

vector::crend

Returns a const iterator that addresses the location succeeding the last element in a reversed vector.

```
const_reverse_iterator crend() const;
```

Return Value

A const reverse random-access iterator that addresses the location succeeding the last element in a reversed vector (the location that had preceded the first element in the unreversed `vector`).

Remarks

`crend` is used with a reversed `vector` just as vector::cend is used with a `vector`.

With the return value of `crend` (suitably decremented), the `vector` object cannot be modified.

`crend` can be used to test to whether a reverse iterator has reached the end of its `vector`.

The value returned by `crend` should not be dereferenced.

Example

```
// vector_crend.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;
   vector <int>::const_reverse_iterator v1_rIter;

   v1.push_back( 1 );
   v1.push_back( 2 );

   for ( v1_rIter = v1.rbegin( ) ; v1_rIter != v1.rend( ) ; v1_rIter++ )
      cout << *v1_rIter << endl;
}
```

Output

```
2
1
```

vector::data

Returns a pointer to the first element in the vector.

```
const_pointer data() const;


pointer data();
```

Return Value

A pointer to the first element in the vector or to the location succeeding an empty `vector`.

Example

```
// vector_data.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main()
{
    using namespace std;
    vector<int> c1;
    vector<int>::pointer c1 ptr;
    vector<int>::const_pointer c1_cPtr;

    c1.push_back(1);
    c1.push_back(2);

    cout << "The vector c1 contains elements:";
    c1_cPtr = c1.data();
    for (size_t n = c1.size(); 0 < n; --n, c1_cPtr++)
    {
        cout << " " << *c1_cPtr;
    }
    cout << endl;

    cout << "The vector c1 now contains elements:";
    c1 ptr = c1.data();
 *c1 ptr = 20;
    for (size_t n = c1.size(); 0 < n; --n, c1 ptr++)
    {
        cout << " " << *c1 ptr;
    }
    cout << endl;
```

```
}
```

Output

```
The vector c1 contains elements: 1 2
The vector c1 now contains elements: 20 2
```

vector::difference_type

A type that provides the difference between two iterators that refer to elements within the same vector.

```
typedef typename Allocator::difference_type difference_type;
```

Remarks

A `difference_type` can also be described as the number of elements between two pointers, because a pointer to an element contains its address.

An iterator is more commonly used to access a vector element.

Example

```
// vector_diff_type.cpp
// compile with: /EHsc
#include <iostream>
#include <vector>
#include <algorithm>

int main( )
{
   using namespace std;

   vector <int> c1;
   vector <int>::iterator c1_Iter, c2_Iter;

   c1.push_back( 30 );
   c1.push_back( 20 );
   c1.push_back( 30 );
   c1.push_back( 10 );
   c1.push_back( 30 );
   c1.push_back( 20 );

   c1_Iter = c1.begin( );
   c2_Iter = c1.end( );
```

```
    vector <int>::difference_type   df_typ1, df_typ2, df_typ3;

    df_typ1 = count( c1_Iter, c2_Iter, 10 );
    df_typ2 = count( c1_Iter, c2_Iter, 20 );
    df_typ3 = count( c1_Iter, c2_Iter, 30 );
    cout << "The number '10' is in c1 collection " << df_typ1 << " times.\n";
    cout << "The number '20' is in c1 collection " << df_typ2 << " times.\n";
    cout << "The number '30' is in c1 collection " << df_typ3 << " times.\n";
}
```

Output

```
The number '10' is in c1 collection 1 times.
The number '20' is in c1 collection 2 times.
The number '30' is in c1 collection 3 times.
```

vector::emplace

Inserts an element constructed in place into the vector at a specified position.

```
iterator emplace(
    const_iterator _Where,
    Type&& val);
```

Parameters

Parameter	Description
`_Where`	The position in the vector where the first element is inserted.
`val`	The value of the element being inserted into the `vector`.

Return Value

The function returns an iterator that points to the position where the new element was inserted into the `vector`.

Remarks

Any insertion operation can be expensive, see vector Class for a discussion of `vector` performance.

Example

```
// vector_emplace.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;
   vector <int>::iterator Iter;

   v1.push_back( 10 );
   v1.push_back( 20 );
   v1.push_back( 30 );

   cout << "v1 =" ;
   for ( Iter = v1.begin( ) ; Iter != v1.end( ) ; Iter++ )
      cout << " " << *Iter;
   cout << endl;

// initialize a vector of vectors by moving v1
   vector < vector <int> > vv1;

   vv1.emplace( vv1.begin(), move( v1 ) );
   if ( vv1.size( ) != 0 && vv1[0].size( ) != 0 )
      {
      cout << "vv1[0] =";
      for (Iter = vv1[0].begin( ); Iter != vv1[0].end( ); Iter++ )
         cout << " " << *Iter;
      cout << endl;
      }
}
```

Output

```
v1 = 10 20 30
vv1[0] = 10 20 30
```

vector::emplace_back

Adds an element constructed in place to the end of the vector.

```
template <class... Types>
void emplace_back(Types&&... _Args);
```

Parameters

Parameter	Description
_Args	Constructor arguments. The function infers which constructor overload to invoke based on the arguments provided.

Example

C++

```
#include <vector>
struct obj
{
   obj(int, double) {}
};

int main()
{
   std::vector<obj> v;
   v.emplace_back(1, 3.14); // obj in created in place in the vector
}
```

vector::empty

Tests if the vector is empty.

```
bool empty() const;
```

Return Value

true if the vector is empty; **false** if the vector is not empty.

Example

```
// vector_empty.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>
```

```
int main( )
{
   using namespace std;
   vector <int> v1;

   v1.push_back( 10 );

   if ( v1.empty( ) )
      cout << "The vector is empty." << endl;
   else
      cout << "The vector is not empty." << endl;
}
```

Output

```
The vector is not empty.
```

vector::end

Returns the past-the-end iterator.

```
iterator end();

const_iterator end() const;
```

Return Value

The past-the-end iterator for the vector. If the vector is empty, `vector::end() == vector::begin()`.

Remarks

If the return value of **end** is assigned to a variable of type `const_iterator`, the vector object cannot be modified. If the return value of **end** is assigned to a variable of type **iterator**, the vector object can be modified.

Example

```
// vector_end.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>
int main( )
{
   using namespace std;
   vector <int> v1;
```

```
    vector <int>::iterator v1_Iter;

    v1.push_back( 1 );
    v1.push_back( 2 );

    for ( v1_Iter = v1.begin( ) ; v1_Iter != v1.end( ) ; v1_Iter++ )
       cout << *v1_Iter << endl;
}
```

Output

```
1
2
```

vector::erase

Removes an element or a range of elements in a vector from specified positions.

```
iterator erase(
    const_iterator _Where);

iterator erase(
    const_iterator first,
    const_iterator last);
```

Parameters

Parameter	Description
`_Where`	Position of the element to be removed from the vector.
`first`	Position of the first element removed from the vector.
`last`	Position just beyond the last element removed from the vector.

Return Value

An iterator that designates the first element remaining beyond any elements removed, or a pointer to the end of the vector if no such element exists.

Example

```
// vector_erase.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;
   vector <int>::iterator Iter;

   v1.push_back( 10 );
   v1.push_back( 20 );
   v1.push_back( 30 );
   v1.push_back( 40 );
   v1.push_back( 50 );

   cout << "v1 =" ;
   for ( Iter = v1.begin( ) ; Iter != v1.end( ) ; Iter++ )
      cout << " " << *Iter;
   cout << endl;

   v1.erase( v1.begin( ) );
   cout << "v1 =";
   for ( Iter = v1.begin( ) ; Iter != v1.end( ) ; Iter++ )
      cout << " " << *Iter;
   cout << endl;

   v1.erase( v1.begin( ) + 1, v1.begin( ) + 3 );
   cout << "v1 =";
   for ( Iter = v1.begin( ) ; Iter != v1.end( ) ; Iter++ )
      cout << " " << *Iter;
   cout << endl;
}
```

Output

```
v1 = 10 20 30 40 50
v1 = 20 30 40 50
v1 = 20 50
```

vector::front

Returns a reference to the first element in a vector.

```
reference front();

const_reference front() const;
```

Return Value

A reference to the first element in the vector object. If the vector is empty, the return is undefined.

Remarks

If the return value of `front` is assigned to a `const_reference`, the vector object cannot be modified. If the return value of `front` is assigned to a **reference**, the vector object can be modified.

When compiled by using `_ITERATOR_DEBUG_LEVEL` defined as 1 or 2, a runtime error will occur if you attempt to access an element in an empty vector. See Checked Iterators for more information.

Example

```
// vector_front.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;

   v1.push_back( 10 );
   v1.push_back( 11 );

   int& i = v1.front( );
   const int& ii = v1.front( );

   cout << "The first integer of v1 is "<< i << endl;
   // by incrementing i, we move the the front reference to the second element
   i++;
   cout << "Now, the first integer of v1 is "<< i << endl;
}
```

vector::get_allocator

Returns a copy of the allocator object used to construct the vector.

```
```

```
Allocator get_allocator() const;
```

Return Value

The allocator used by the vector.

Remarks

Allocators for the vector class specify how the class manages storage. The default allocators supplied with STL container classes are sufficient for most programming needs. Writing and using your own allocator class is an advanced C++ topic.

Example

```
// vector_get_allocator.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   // The following lines declare objects that use the default allocator.
   vector<int> v1;
   vector<int, allocator<int> > v2 = vector<int, allocator<int> >(allocator<int>(
)) ;

   // v3 will use the same allocator class as v1
   vector <int> v3( v1.get_allocator( ) );

   vector<int>::allocator_type xvec = v3.get_allocator( );
   // You can now call functions on the allocator class used by vec
}
```

vector::insert

Inserts an element or a number of elements or a range of elements into the vector at a specified position.

```
iterator insert(
    const_iterator _Where,
    const Type& val);

iterator insert(
    const_iterator _Where,
    Type&& val);
```

```
void insert(
    const_iterator _Where,
    size_type count,
    const Type& val);

template <class InputIterator>
void insert(
    const_iterator _Where,
    InputIterator first,
    InputIterator last);
```

Parameters

Parameter	Description
`_Where`	The position in the vector where the first element is inserted.
`val`	The value of the element being inserted into the vector.
`count`	The number of elements being inserted into the vector.
`first`	The position of the first element in the range of elements to be copied.
`last`	The position of the first element beyond the range of elements to be copied.

Return Value

The first two `insert` functions return an iterator that points to the position where the new element was inserted into the vector.

Remarks

As a precondition, `first` and `last` must not be iterators into the vector, or the behavior is undefined. Any insertion operation can be expensive, see vector Class for a discussion of `vector` performance.

Example

```
// vector_insert.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;
```

```
   vector <int>::iterator Iter;

   v1.push_back( 10 );
   v1.push_back( 20 );
   v1.push_back( 30 );

   cout << "v1 =" ;
   for ( Iter = v1.begin( ) ; Iter != v1.end( ) ; Iter++ )
      cout << " " << *Iter;
   cout << endl;

   v1.insert( v1.begin( ) + 1, 40 );
   cout << "v1 =";
   for ( Iter = v1.begin( ) ; Iter != v1.end( ) ; Iter++ )
      cout << " " << *Iter;
   cout << endl;
   v1.insert( v1.begin( ) + 2, 4, 50 );

   cout << "v1 =";
   for ( Iter = v1.begin( ) ; Iter != v1.end( ) ; Iter++ )
      cout << " " << *Iter;
   cout << endl;

   v1.insert( v1.begin( )+1, v1.begin( )+2, v1.begin( )+4 );
   cout << "v1 =";
   for (Iter = v1.begin( ); Iter != v1.end( ); Iter++ )
      cout << " " << *Iter;
   cout << endl;

// initialize a vector of vectors by moving v1
   vector < vector <int> > vv1;

   vv1.insert( vv1.begin(), move( v1 ) );
   if ( vv1.size( ) != 0 && vv1[0].size( ) != 0 )
      {
      vector < vector <int> >::iterator Iter;
      cout << "vv1[0] =";
      for (Iter = vv1[0].begin( ); Iter != vv1[0].end( ); Iter++ )
         cout << " " << *Iter;
      cout << endl;
      }
}
```

Output

```
v1 = 10 20 30
v1 = 10 40 20 30
v1 = 10 40 50 50 50 50 20 30
v1 = 10 50 50 40 50 50 50 50 20 30
vv1[0] = 10 50 50 40 50 50 50 50 20 30
```

vector::iterator

A type that provides a random-access iterator that can read or modify any element in a vector.

```
typedef implementation-defined iterator;
```

Remarks

A type **iterator** can be used to modify the value of an element.

Example

See the example for begin.

vector::max_size

Returns the maximum length of the vector.

```
size_type max_size() const;
```

Return Value

The maximum possible length of the vector.

Example

```
// vector_max_size.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;
   vector <int>::size_type i;

   i = v1.max_size( );
   cout << "The maximum possible length of the vector is " << i << "." << endl;
}
```

vector::operator[]

Returns a reference to the vector element at a specified position.

```
reference operator[](size_type Pos);

const_reference operator[](size_type Pos) const;
```

Parameters

Parameter	Description
Pos	The position of the vector element.

Return Value

If the position specified is greater than or equal to the size of the container, the result is undefined.

Remarks

If the return value of `operator[]` is assigned to a `const_reference`, the vector object cannot be modified. If the return value of `operator[]` is assigned to a reference, the vector object can be modified.

When compiling with _SECURE_SCL 1 (controlled with _ITERATOR_DEBUG_LEVEL), a runtime error will occur if you attempt to access an element outside the bounds of the vector. See Checked Iterators for more information.

Example

C++

```
// vector_op_ref.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;

   v1.push_back( 10 );
   v1.push_back( 20 );

   int& i = v1[1];
   cout << "The second integer of v1 is " << i << endl;
```

```
}
```

vector::operator=

Replaces the elements of the vector with a copy of another vector.

```
vector& operator=(const vector& right);

vector& operator=(vector&& right);
```

Parameters

Parameter	Description
`right`	The vector being copied into the `vector`.

Remarks

After erasing any existing elements in a `vector`, `operator=` either copies or moves the contents of `right` into the `vector`.

Example

```
// vector_operator_as.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector<int> v1, v2, v3;
   vector<int>::iterator iter;

   v1.push_back(10);
   v1.push_back(20);
   v1.push_back(30);
   v1.push_back(40);
   v1.push_back(50);

   cout << "v1 = " ;
   for (iter = v1.begin(); iter != v1.end(); iter++)
```

```
      cout << *iter << " ";
   cout << endl;

   v2 = v1;
   cout << "v2 = ";
   for (iter = v2.begin(); iter != v2.end(); iter++)
      cout << *iter << " ";
   cout << endl;

// move v1 into v2
   v2.clear();
   v2 = move(v1);
   cout << "v2 = ";
   for (iter = v2.begin(); iter != v2.end(); iter++)
      cout << *iter << " ";
   cout << endl;
}
```

vector::pointer

A type that provides a pointer to an element in a vector.

```
typedef typename Allocator::pointer pointer;
```

Remarks

A type **pointer** can be used to modify the value of an element.

Example

```
// vector_pointer.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector<int> v;
   v.push_back( 11 );
   v.push_back( 22 );

   vector<int>::pointer ptr = &v[0];
   cout << *ptr << endl;
   ptr++;
   cout << *ptr << endl;
```

```
 *ptr = 44;
   cout << *ptr << endl;
}
```

Output

```
11
22
44
```

vector::pop_back

Deletes the element at the end of the vector.

```
void pop_back();
```

Remarks

For a code example, see vector::push_back().

vector::push_back

Adds an element to the end of the vector.

```
void push_back(const T& Val);


void push_back(T&& Val);
```

Parameters

`Val`
The value to assign to the element added to the end of the vector.

Example

C++

```
// compile with: /EHsc /W4
#include <vector>
```

```
#include <iostream>

using namespace std;

template <typename T> void print_elem(const T& t) {
    cout << "(" << t << ") ";
}

template <typename T> void print_collection(const T& t) {
    cout << "  " << t.size() << " elements: ";

    for (const auto& p : t) {
        print_elem(p);
    }
    cout << endl;
}

int main()
{
    vector<int> v;
    for (int i = 0; i < 10; ++i) {
        v.push_back(10 + i);
    }

    cout << "vector data: " << endl;
    print_collection(v);

    // pop_back() until it's empty, printing the last element as we go
    while (v.begin() != v.end()) {
        cout << "v.back(): "; print_elem(v.back()); cout << endl;
        v.pop_back();
    }
}
```

vector::rbegin

Returns an iterator to the first element in a reversed vector.

```
reverse_iterator rbegin();

const_reverse_iterator rbegin() const;
```

Return Value

A reverse random-access iterator addressing the first element in a reversed vector or addressing what had been the last element in the unreversed vector.

Remarks

If the return value of `rbegin` is assigned to a `const_reverse_iterator`, the vector object cannot be modified. If the return value of `rbegin` is assigned to a `reverse_iterator`, the vector object can be modified.

Example

```
// vector_rbegin.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;
   vector <int>::iterator v1_Iter;
   vector <int>::reverse_iterator v1_rIter;

   v1.push_back( 1 );
   v1.push_back( 2 );

   v1_Iter = v1.begin( );
   cout << "The first element of vector is "
        << *v1_Iter << "." << endl;

   v1_rIter = v1.rbegin( );
   cout << "The first element of the reversed vector is "
        << *v1_rIter << "." << endl;
}
```

Output

```
The first element of vector is 1.
The first element of the reversed vector is 2.
```

vector::reference

A type that provides a reference to an element stored in a vector.

```
typedef typename Allocator::reference reference;
```

Example

See at for an example of how to use **reference** in the vector class.

vector::rend

Returns an iterator that addresses the location succeeding the last element in a reversed vector.

```
const_reverse_iterator rend() const;

reverse_iterator rend();
```

Return Value

A reverse random-access iterator that addresses the location succeeding the last element in a reversed vector (the location that had preceded the first element in the unreversed vector).

Remarks

rend is used with a reversed vector just as end is used with a vector.

If the return value of rend is assigned to a `const_reverse_iterator`, then the vector object cannot be modified. If the return value of rend is assigned to a `reverse_iterator`, then the vector object can be modified.

rend can be used to test to whether a reverse iterator has reached the end of its vector.

The value returned by rend should not be dereferenced.

Example

```
// vector_rend.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;
   vector <int>::reverse_iterator v1_rIter;

   v1.push_back( 1 );
   v1.push_back( 2 );

   for ( v1_rIter = v1.rbegin( ) ; v1_rIter != v1.rend( ) ; v1_rIter++ )
      cout << *v1_rIter << endl;
}
```

Output

```
2
1
```

vector::reserve

Reserves a minimum length of storage for a vector object, allocating space if necessary.

```
void reserve(size_type count);
```

Parameters

count
The minimum length of storage to be allocated for the vector.

Example

```
// vector_reserve.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;
   //vector <int>::iterator Iter;

   v1.push_back( 1 );
   cout << "Current capacity of v1 = "
      << v1.capacity( ) << endl;
   v1.reserve( 20 );
   cout << "Current capacity of v1 = "
      << v1.capacity( ) << endl;
}
```

Output

```
Current capacity of v1 = 1
Current capacity of v1 = 20
```

vector::resize

Specifies a new size for a vector.

```
void resize(size_type Newsize);

void resize(size_type Newsize, Type Val);
```

Parameters

`Newsize`
The new size of the vector.

`Val`
The initialization value of new elements added to the vector if the new size is larger that the original size. If the value is omitted, the new objects use their default constructor.

Remarks

If the container's size is less than the requested size, `Newsize`, elements are added to the vector until it reaches the requested size. If the container's size is larger than the requested size, the elements closest to the end of the container are deleted until the container reaches the size `Newsize`. If the present size of the container is the same as the requested size, no action is taken.

size reflects the current size of the vector.

Example

C++

```
// vectorsizing.cpp
// compile with: /EHsc /W4
// Illustrates vector::reserve, vector::max_size,
// vector::resize, vector::resize, and vector::capacity.
//
// Functions:
//
//    vector::max_size - Returns maximum number of elements vector could
//                       hold.
//
//    vector::capacity - Returns number of elements for which memory has
//                       been allocated.
//
//    vector::size - Returns number of elements in the vector.
//
//    vector::resize - Reallocates memory for vector, preserves its
//                     contents if new size is larger than existing size.
//
//    vector::reserve - Allocates elements for vector to ensure a minimum
//                      size, preserving its contents if the new size is
```

```
//                      larger than existing size.
//
//    vector::push_back - Appends (inserts) an element to the end of a
//                        vector, allocating memory for it if necessary.
//
////////////////////////////////////////////////////////////////////////

// The debugger cannot handle symbols more than 255 characters long.
// STL often creates symbols longer than that.
// The warning can be disabled:
//#pragma warning(disable:4786)

#include <iostream>
#include <vector>
#include <string>

using namespace std;

template <typename C> void print(const string& s, const C& c) {
    cout << s;

    for (const auto& e : c) {
        cout << e << " ";
    }
    cout << endl;
}

void printvstats(const vector<int>& v) {
    cout << "   the vector's size is: " << v.size() << endl;
    cout << "   the vector's capacity is: " << v.capacity() << endl;
    cout << "   the vector's maximum size is: " << v.max_size() << endl;
}

int main()
{
    // declare a vector that begins with 0 elements.
    vector<int> v;

    // Show statistics about vector.
    cout << endl << "After declaring an empty vector:" << endl;
    printvstats(v);
    print("   the vector's contents: ", v);

    // Add one element to the end of the vector.
    v.push_back(-1);
    cout << endl << "After adding an element:" << endl;
    printvstats(v);
    print("   the vector's contents: ", v);

    for (int i = 1; i < 10; ++i) {
        v.push_back(i);
    }
    cout << endl << "After adding 10 elements:" << endl;
    printvstats(v);
    print("   the vector's contents: ", v);
```

```
    v.resize(6);
    cout << endl << "After resizing to 6 elements without an initialization
value:" << endl;
    printvstats(v);
    print("   the vector's contents: ", v);

    v.resize(9, 999);
    cout << endl << "After resizing to 9 elements with an initialization value of
999:" << endl;
    printvstats(v);
    print("   the vector's contents: ", v);

    v.resize(12);
    cout << endl << "After resizing to 12 elements without an initialization
value:" << endl;
    printvstats(v);
    print("   the vector's contents: ", v);

    // Ensure there's room for at least 1000 elements.
    v.reserve(1000);
    cout << endl << "After vector::reserve(1000):" << endl;
    printvstats(v);

    // Ensure there's room for at least 2000 elements.
    v.resize(2000);
    cout << endl << "After vector::resize(2000):" << endl;
    printvstats(v);
}
```

vector::reverse_iterator

A type that provides a random-access iterator that can read or modify any element in a reversed vector.

```
typedef std::reverse_iterator<iterator> reverse_iterator;
```

Remarks

A type `reverse_iterator` is used to iterate through the vector in reverse.

Example

See the example for rbegin.

vector::shrink_to_fit

Discards excess capacity.

```
void shrink_to_fit();
```

Example

```
// vector_shrink_to_fit.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;
   //vector <int>::iterator Iter;

   v1.push_back( 1 );
   cout << "Current capacity of v1 = "
      << v1.capacity( ) << endl;
   v1.reserve( 20 );
   cout << "Current capacity of v1 = "
      << v1.capacity( ) << endl;
   v1.shrink_to_fit();
   cout << "Current capacity of v1 = "
      << v1.capacity( ) << endl;
}
```

Output

```
Current capacity of v1 = 1
Current capacity of v1 = 20
Current capacity of v1 = 1
```

vector::size

Returns the number of elements in the vector.

```
size_type size() const;
```

Return Value

The current length of the vector.

Example

```
// vector_size.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1;
   vector <int>::size_type i;

   v1.push_back( 1 );
   i = v1.size( );
   cout << "Vector length is " << i << "." << endl;

   v1.push_back( 2 );
   i = v1.size( );
   cout << "Vector length is now " << i << "." << endl;
}
```

Output

```
Vector length is 1.
Vector length is now 2.
```

vector::size_type

A type that counts the number of elements in a vector.

```
typedef typename Allocator::size_type size_type;
```

Example

See the example for capacity.

vector::swap

Exchanges the elements of two vectors.

```
void swap(
    vector<Type, Allocator>& right);

friend void swap(
    vector<Type, Allocator>& left,
    vector<Type, Allocator>& right);
```

Parameters

`right`
A vector providing the elements to be swapped, or a vector whose elements are to be exchanged with those of the vector `left`.

`left`
A vector whose elements are to be exchanged with those of the vector `right`.

Example

```
// vector_swap.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector <int> v1, v2;

   v1.push_back( 1 );
   v1.push_back( 2 );
   v1.push_back( 3 );

   v2.push_back( 10 );
   v2.push_back( 20 );

   cout << "The number of elements in v1 = " << v1.size( ) << endl;
   cout << "The number of elements in v2 = " << v2.size( ) << endl;
   cout << endl;

   v1.swap( v2 );

   cout << "The number of elements in v1 = " << v1.size( ) << endl;
   cout << "The number of elements in v2 = " << v2.size( ) << endl;
}
```

Output

```
The number of elements in v1 = 3
The number of elements in v2 = 2

The number of elements in v1 = 2
The number of elements in v2 = 3
```

vector::value_type

A type that represents the data type stored in a vector.

```
typedef typename Allocator::value_type value_type;
```

Remarks

`value_type` is a synonym for the template parameter **Type**.

Example

```
// vector_value_type.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main( )
{
   using namespace std;
   vector<int>::value_type AnInt;
   AnInt = 44;
   cout << AnInt << endl;
}
```

Output

```
44
```

vector::vector

Constructs a vector of a specific size or with elements of a specific value or with a specific allocator or as a copy of all or part of some other vector.

```
vector();

explicit vector(
    const Allocator& Al);

explicit vector(
    size_type Count);

vector(
    size_type Count,
    const Type& Val);

vector(
    size_type Count,
    const Type& Val,
    const Allocator& Al);

vector(
    const vector& Right);

vector(
    vector&& Right);

vector(
    initializer_list<Type> IList,
    const _Allocator& Al);

template <class InputIterator>
vector(
 InputIterator First,
    InputIterator Last);

template <class InputIterator>
vector(
 InputIterator First,
    InputIterator Last,
    const Allocator& Al);
```

Parameters

Parameter	Description
`Al`	The allocator class to use with this object. get_allocator returns the allocator class for the object.
`Count`	The number of elements in the constructed vector.
`Val`	The value of the elements in the constructed vector.

`Right`	The vector of which the constructed vector is to be a copy.
`First`	Position of the first element in the range of elements to be copied.
`Last`	Position of the first element beyond the range of elements to be copied.
`IList`	The initializer_list containing the elmeents to copy.

Remarks

All constructors store an allocator object (`Al`) and initialize the vector.

The first two constructors specify an empty initial vector. The second explicitly specifies the allocator type (`Al`) to be used.

The third constructor specifies a repetition of a specified number (`Count`) of elements of the default value for class `Type`.

The fourth and fifth constructors specify a repetition of (`Count`) elements of value `Val`.

The sixth constructor specifies a copy of the vector `Right`.

The seventh constructor moves the vector `Right`.

The eighth constructor uses an initializer_list to specify the elements.

The ninth and tenth constructors copy the range [`First`, `Last`) of a vector.

Example

C++

```
// vector_ctor.cpp
// compile with: /EHsc
#include <vector>
#include <iostream>

int main()
{
    using namespace std;
    vector <int>::iterator v1_Iter, v2_Iter, v3_Iter, v4_Iter, v5_Iter, v6_Iter;

    // Create an empty vector v0
    vector <int> v0;

    // Create a vector v1 with 3 elements of default value 0
    vector <int> v1(3);

    // Create a vector v2 with 5 elements of value 2
    vector <int> v2(5, 2);

    // Create a vector v3 with 3 elements of value 1 and with the allocator
```

```
    // of vector v2
    vector <int> v3(3, 1, v2.get_allocator());

    // Create a copy, vector v4, of vector v2
    vector <int> v4(v2);

    // Create a new temporary vector for demonstrating copying ranges
    vector <int> v5(5);
    for (auto i : v5) {
        v5[i] = i;
    }

    // Create a vector v6 by copying the range v5[ first,  last)
    vector <int> v6(v5.begin() + 1, v5.begin() + 3);

    cout << "v1 =";
    for (auto& v : v1){
        cout << " " << v;
    }
    cout << endl;

    cout << "v2 =";
    for (auto& v : v2){
        cout << " " << v;
    }
    cout << endl;

    cout << "v3 =";
    for (auto& v : v3){
        cout << " " << v;
    }
    cout << endl;
    cout << "v4 =";
    for (auto& v : v4){
        cout << " " << v;
    }
    cout << endl;

    cout << "v5 =";
    for (auto& v : v5){
        cout << " " << v;
    }
    cout << endl;

    cout << "v6 =";
    for (auto& v : v6){
        cout << " " << v;
    }
    cout << endl;

    // Move vector v2 to vector v7
    vector <int> v7(move(v2));
    vector <int>::iterator v7_Iter;

    cout << "v7 =";
```

```
    for (auto& v : v7){
        cout << " " << v;
    }
    cout << endl;

    vector<int> v8{ { 1, 2, 3, 4 } };
    for (auto& v : v8){
        cout << " " << v ;
    }
    cout << endl;
}
```

Output

```
v1 = 0 0 0v2 = 2 2 2 2 2v3 = 1 1 1v4 = 2 2 2 2 2v5 = 0 1 2 3 4v6 = 1 2v7 = 2 2 2 2
21 2 3 4
```

See Also

Thread Safety in the C++ Standard Library
Standard Template Library