deque 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 deque Class.

Arranges elements of a given type in a linear arrangement and, like a vector, enables fast random access to any element, and efficient insertion and deletion at the back of the container. However, unlike a vector, the deque class also supports efficient insertion and deletion at the front of the container.

Syntax

unstlib

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

Parameters

`Type`
The element data type to be stored in the deque.

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

Remarks

The choice of container type should be based in general on the type of searching and inserting required by the application. Vectors should be the preferred container for managing a sequence when random access to any element is at a premium and insertions or deletions of elements are only required at the end of a sequence. The performance of the list container is superior when efficient insertions and deletions (in constant time) at any location within the sequence is at a premium. Such operations in the middle of the sequence require element copies and assignments proportional to the number of elements in the sequence (linear time).

Deque reallocation occurs when a member function must insert or erase elements of the sequence:

- If an element is inserted into an empty sequence, or if an element is erased to leave an empty sequence, then iterators earlier returned by begin and end become invalid.
- If an element is inserted at the first position of the deque, then all iterators, but no references, that designate existing elements become invalid.

- If an element is inserted at the end of the deque, then end and all iterators, but no references, that designate existing elements become invalid.
- If an element is erased at the front of the deque, only that iterator and references to the erased element become invalid.
- If the last element is erased from the end of the deque, only that iterator to the final element and references to the erased element become invalid.

Otherwise, inserting or erasing an element invalidates all iterators and references.

Constructors

deque	Constructs a `deque`. Several constructors are provided to set up the contents of the new `deque` in different ways: empty; loaded with a specified number of empty elements; contents moved or copied from another `deque`; contents copied or moved by using an iterator; and one element copied into the `deque`` `count` times. Some of the constructors enable using a custom `allocator` to create elements.

Typedefs

allocator_type	A type that represents the `allocator` class for the `deque` object.
const_iterator	A type that provides a random-access iterator that can access and read elements in the `deque` as `const`
const_pointer	A type that provides a pointer to an element in a `deque` as a `const`.
const_reference	A type that provides a reference to an element in a `deque` for reading and other operations as a `const`.
const_reverse_iterator	A type that provides a random-access iterator that can access and read elements in the `deque` as `const`. The deque is viewed in reverse. For more information, see reverse_iterator Class
difference_type	A type that provides the difference between two random-access iterators that refer to elements in the same `deque`.
iterator	A type that provides a random-access iterator that can read or modify any element in a `deque`.
pointer	A type that provides a pointer to an element in a `deque`.
reference	A type that provides a reference to an element stored in a `deque`.
reverse_iterator	

	A type that provides a random-access iterator that can read or modify an element in a deque. The deque is viewed in reverse order.
size_type	A type that counts the number of elements in a deque.
value_type	A type that represents the data type stored in a deque.

Member Functions

assign	Erases elements from a deque and copies a new sequence of elements to the target deque.
at	Returns a reference to the element at a specified location in the deque.
back	Returns a reference to the last element of the deque.
begin	Returns a random-access iterator addressing the first element in the deque.
deque::cbegin	Returns a const iterator to the first element in the deque.
deque::cend	Returns a random-access `const` iterator that points just beyond the end of the deque.
clear	Erases all the elements of a deque.
deque::crbegin	Returns a random-access const iterator to the first element in a deque viewed in reverse order.
deque::crend	Returns a random-access const iterator to the first element in a deque viewed in reverse order.
deque::emplace	Inserts an element constructed in place into the deque at a specified position.
deque::emplace_back	Adds an element constructed in place to the end of the deque.
deque::emplace_front	Adds an element constructed in place to the start of the deque.
empty	Returns `true` if the deque contains zero elements, and `false` if it contains one or more elements.
end	Returns a random-access iterator that points just beyond the end of the deque.
erase	Removes an element or a range of elements in a deque from specified positions.
front	Returns a reference to the first element in a deque.

get_allocator	Returns a copy of the `allocator` object that is used to construct the deque.
insert	Inserts an element, several elements, or a range of elements into the deque at a specified position.
max_size	Returns the maximum possible length of the deque.
pop_back	Erases the element at the end of the deque.
pop_front	Erases the element at the start of the deque.
push_back	Adds an element to the end of the deque.
push_front	Adds an element to the start of the deque.
rbegin	Returns a random-access iterator to the first element in a reversed deque.
rend	Returns a random-access iterator that points just beyond the last element in a reversed deque.
resize	Specifies a new size for a deque.
deque::shrink_to_fit	Discards excess capacity.
size	Returns the number of elements in the deque.
swap	Exchanges the elements of two deques.

Operators

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

Requirements

Header: <deque>

deque::allocator_type

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

```
typedef Allocator allocator_type;
```

Remarks

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

Example

See the example for get_allocator.

deque::assign

Erases elements from a deque and copies a new set of elements to the target deque.

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

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

void assign(
    initializer_list<Type>
IList);
```

Parameters

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

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

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

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

`IList`
The initializer_list being inserted into the deque.

Remarks

After any existing elements in the target deque are erased, `assign` either inserts a specified range of elements from the original deque or from some other deque into the target deque, or inserts copies of a new element of a specified value into the target deque.

Example

```
// deque_assign.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>
#include <initializer_list>

int main()
{
    using namespace std;
    deque <int> c1, c2;
    deque <int>::const_iterator cIter;

    c1.push_back(10);
    c1.push_back(20);
    c1.push_back(30);
    c2.push_back(40);
    c2.push_back(50);
    c2.push_back(60);

    deque<int> d1{ 1, 2, 3, 4 };
    initializer_list<int> iList{ 5, 6, 7, 8 };
    d1.assign(iList);

    cout << "d1 = ";
    for (int i : d1)
        cout << i;
    cout << endl;

    cout << "c1 =";
    for (int i : c1)
        cout << i;
    cout << endl;

    c1.assign(++c2.begin(), c2.end());
    cout << "c1 =";
    for (int i : c1)
        cout << i;
    cout << endl;

    c1.assign(7, 4);
    cout << "c1 =";
    for (int i : c1)
        cout << i;
    cout << endl;
```

```
}
```

Output

```
d1 = 5678c1 =102030c1 =5060c1 =4444444
```

deque::at

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

```
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 deque.

Return Value

If `_Pos` is greater than the size of the deque, **at** throws an exception.

Return Value

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

Example

```
// deque_at.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1;

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

```
   const int& i = c1.at( 0 );
   int& j = c1.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
```

deque::back

Returns a reference to the last element of the deque.

```
reference back();

const_reference back() const;
```

Return Value

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

Remarks

If the return value of **back** is assigned to a `const_reference`, the deque object cannot be modified. If the return value of **back** is assigned to a **reference**, the deque 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 deque. See Checked Iterators for more information.

Example

```
// deque_back.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1;

   c1.push_back( 10 );
```

```
   c1.push_back( 11 );

   int& i = c1.back( );
   const int& ii = c1.front( );

   cout << "The last integer of c1 is " << i << endl;
   i--;
   cout << "The next-to-last integer of c1 is " << ii << endl;
}
```

Output

```
The last integer of c1 is 11
The next-to-last integer of c1 is 10
```

deque::begin

Returns an iterator addressing the first element in the deque.

```
const_iterator begin() const;

iterator begin();
```

Return Value

A random-access iterator addressing the first element in the deque or to the location succeeding an empty deque.

Remarks

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

Example

```
// deque_begin.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1;
```

```
    deque <int>::iterator c1_Iter;
    deque <int>::const_iterator c1_cIter;

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

    c1_Iter = c1.begin( );
    cout << "The first element of c1 is " << *c1_Iter << endl;

 *c1_Iter = 20;
    c1_Iter = c1.begin( );
    cout << "The first element of c1 is now " << *c1_Iter << endl;

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

Output

```
The first element of c1 is 1
The first element of c1 is now 20
```

deque::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
```

deque::cend

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

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

Return Value

A 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.

deque::clear

Erases all the elements of a deque.

```
void clear();
```

Example

```
// deque_clear.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1;

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

   cout << "The size of the deque is initially " << c1.size( ) << endl;
   c1.clear( );
   cout << "The size of the deque after clearing is " << c1.size( ) << endl;
}
```

Output

```
The size of the deque is initially 3
The size of the deque after clearing is 0
```

deque::const_iterator

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

```
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.

deque::const_pointer

Provides a pointer to a const element in a deque.

unstlib

```
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 deque element.

deque::const_reference

A type that provides a reference to a **const** element stored in a deque 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

```
// deque_const_ref.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1;

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

   const deque <int> c2 = c1;
   const int &i = c2.front( );
   const int &j = c2.back( );
   cout << "The first element is " << i << endl;
   cout << "The second element is " << j << endl;

   // The following line would cause an error as c2 is const
   // c2.push_back( 30 );
```

```
}
```

Output

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

deque::const_reverse_iterator

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

```
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 deque in reverse.

Example

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

deque::crbegin

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

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

Return Value

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

Remarks

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

Example

```
// deque_crbegin.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

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

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

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

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

Output

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

deque::crend

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

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

Return Value

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

Remarks

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

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

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

The value returned by crend should not be dereferenced.

Example

```
// deque_crend.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

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

deque::deque

Constructs a deque 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 deque.

```
deque();

explicit deque(
    const Allocator& Al);

explicit deque(
    size_type Count);

deque(
    size_type Count,
    const Type& Val);
```

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

deque(
    const deque& Right);

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

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

deque(
    initializer_list<value_type>
IList,
    const Allocator& Al);
```

Parameters

Parameter	Description
`Al`	The allocator class to use with this object.
`Count`	The number of elements in the constructed deque.
`Val`	The value of the elements in the constructed deque.
`Right`	The deque of which the constructed deque 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 to be copied.

Remarks

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

The first two constructors specify an empty initial deque; the second one also 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 deque `Right`.

The seventh and eighth constructors copy the range `[First, Last)` of a deque.

The seventh constructor moves the deque `Right`.

The eighth constructor copies the contents of an initializer_list.

None of the constructors perform any interim reallocations.

Example

```
/ compile with: /EHsc
#include <deque>
#include <iostream>
#include <forward_list>

int main()
{
    using namespace std;

    forward_list<int> f1{ 1, 2, 3, 4 };

    f1.insert_after(f1.begin(), { 5, 6, 7, 8 });

    deque <int>::iterator c1_Iter, c2_Iter, c3_Iter, c4_Iter, c5_Iter, c6_Iter;

    // Create an empty deque c0
    deque <int> c0;

    // Create a deque c1 with 3 elements of default value 0
    deque <int> c1(3);

    // Create a deque c2 with 5 elements of value 2
    deque <int> c2(5, 2);

    // Create a deque c3 with 3 elements of value 1 and with the
    // allocator of deque c2
    deque <int> c3(3, 1, c2.get_allocator());

    // Create a copy, deque c4, of deque c2
    deque <int> c4(c2);

    // Create a deque c5 by copying the range c4[ first,  last)
    c4_Iter = c4.begin();
    c4_Iter++;
```

```
c4_Iter++;
deque <int> c5(c4.begin(), c4_Iter);

// Create a deque c6 by copying the range c4[ first,  last) and
// c2 with the allocator of deque
c4_Iter = c4.begin();
c4_Iter++;
c4_Iter++;
c4_Iter++;
deque <int> c6(c4.begin(), c4_Iter, c2.get_allocator());

// Create a deque c8 by copying the contents of an initializer_list
// using brace initialization
deque<int> c8({ 1, 2, 3, 4 });

initializer_list<int> iList{ 5, 6, 7, 8 };
deque<int> c9( iList);

cout << "c1 = ";
for (int i : c1)
    cout << i << " ";
cout << endl;

cout << "c2 = ";
for (int i : c2)
    cout << i << " ";
cout << endl;

cout << "c3 = ";
for (int i : c3)
    cout << i << " ";
cout << endl;

cout << "c4 = ";
for (int i : c4)
    cout << i << " ";
cout << endl;

cout << "c5 = ";
for (int i : c5)
    cout << i << " ";
cout << endl;

cout << "c6 = ";
for (int i : c6)
    cout << i << " ";
cout << endl;

// Move deque c6 to deque c7
deque <int> c7(move(c6));
deque <int>::iterator c7_Iter;

cout << "c7 =";
for (int i : c7)
    cout << i << " ";
```

```
    cout << endl;

    cout << "c8 = ";
    for (int i : c8)
        cout << i << " ";
    cout << endl;

    cout << "c9 = ";
    for (int i : c9)
        cout << i << " ";
    cout << endl;

    int x = 3;
}
// deque_deque.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
    using namespace std;
   deque <int>::iterator c1_Iter, c2_Iter, c3_Iter, c4_Iter, c5_Iter, c6_Iter;

    // Create an empty deque c0
    deque <int> c0;

    // Create a deque c1 with 3 elements of default value 0
    deque <int> c1( 3 );

    // Create a deque c2 with 5 elements of value 2
    deque <int> c2( 5, 2 );

    // Create a deque c3 with 3 elements of value 1 and with the
    // allocator of deque c2
    deque <int> c3( 3, 1, c2.get_allocator( ) );

    // Create a copy, deque c4, of deque c2
    deque <int> c4( c2 );

    // Create a deque c5 by copying the range c4[ first,  last)
    c4_Iter = c4.begin( );
    c4_Iter++;
    c4_Iter++;
    deque <int> c5( c4.begin( ), c4_Iter );

    // Create a deque c6 by copying the range c4[ first,  last) and
    // c2 with the allocator of deque
    c4_Iter = c4.begin( );
   c4_Iter++;
   c4_Iter++;
   c4_Iter++;
   deque <int> c6( c4.begin( ), c4_Iter, c2.get_allocator( ) );

    // Create a deque c8 by copying the contents of an initializer_list
```

```
// using brace initialization
deque<int> c8({ 1, 2, 3, 4 });

    initializer_list<int> iList{ 5, 6, 7, 8 };
deque<int> c9( iList);

cout << "c1 = ";
for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
    cout << *c1_Iter << " ";
cout << endl;

cout << "c2 = ";
for ( c2_Iter = c2.begin( ); c2_Iter != c2.end( ); c2_Iter++ )
    cout << *c2_Iter << " ";
cout << endl;

cout << "c3 = ";
for ( c3_Iter = c3.begin( ); c3_Iter != c3.end( ); c3_Iter++ )
    cout << *c3_Iter << " ";
cout << endl;

cout << "c4 = ";
for ( c4_Iter = c4.begin( ); c4_Iter != c4.end( ); c4_Iter++ )
    cout << *c4_Iter << " ";
cout << endl;

cout << "c5 = ";
for ( c5_Iter = c5.begin( ); c5_Iter != c5.end( ); c5_Iter++ )
    cout << *c5_Iter << " ";
cout << endl;

cout << "c6 = ";
for ( c6_Iter = c6.begin( ); c6_Iter != c6.end( ); c6_Iter++ )
    cout << *c6_Iter << " ";
cout << endl;

// Move deque c6 to deque c7
deque <int> c7( move(c6) );
deque <int>::iterator c7_Iter;

cout << "c7 =" ;
for ( c7_Iter = c7.begin( ) ; c7_Iter != c7.end( ) ; c7_Iter++ )
    cout << " " << *c7_Iter;
cout << endl;

cout << "c8 = ";
for (int i : c8)
    cout << i << " ";
cout << endl;

cout << "c9 = ";
or (int i : c9)
    cout << i << " ";
cout << endl;
```

```
}
```

deque::difference_type

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

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

Remarks

A `difference_type` can also be described as the number of elements between two pointers.

Example

```
// deque_diff_type.cpp
// compile with: /EHsc
#include <iostream>
#include <deque>
#include <algorithm>

int main( )
{
   using namespace std;

   deque <int> c1;
   deque <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( );

   deque <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.
```

deque::emplace

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

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

Parameters

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

Return Value

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

Remarks

Any insertion operation can be expensive, see deque for a discussion of deque performance.

Example

```
// deque_emplace.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
```

```
{
   using namespace std;
   deque <int> v1;
   deque <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 deque of deques by moving v1
   deque < deque <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
```

deque::emplace_back

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

```
void emplace_back(Type&& val);
```

Parameters

Parameter	Description
`val`	The element added to the end of the deque.

Example

```
// deque_emplace_back.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

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

   v1.push_back( 1 );
   if ( v1.size( ) != 0 )
      cout << "Last element: " << v1.back( ) << endl;

   v1.push_back( 2 );
   if ( v1.size( ) != 0 )
      cout << "New last element: " << v1.back( ) << endl;

// initialize a deque of deques by moving v1
   deque < deque <int> > vv1;

   vv1.emplace_back( move( v1 ) );
   if ( vv1.size( ) != 0 && vv1[0].size( ) != 0 )
      cout << "Moved last element: " << vv1[0].back( ) << endl;
}
```

Output

```
Last element: 1
New last element: 2
Moved last element: 2
```

deque::emplace_front

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

```
void emplace_front(Type&& val);
```

Parameters

Parameter	Description
`val`	The element added to the beginning of the deque.

Example

```
// deque_emplace_front.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

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

   v1.push_back( 1 );
   if ( v1.size( ) != 0 )
      cout << "Last element: " << v1.back( ) << endl;

   v1.push_back( 2 );
   if ( v1.size( ) != 0 )
      cout << "New last element: " << v1.back( ) << endl;

// initialize a deque of deques by moving v1
   deque < deque <int> > vv1;

   vv1.emplace_front( move( v1 ) );
   if ( vv1.size( ) != 0 && vv1[0].size( ) != 0 )
      cout << "Moved last element: " << vv1[0].back( ) << endl;
}
```

Output

```
Last element: 1
New last element: 2
Moved last element: 2
```

deque::empty

Tests if a deque is empty.

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

Return Value

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

Example

```
// deque_empty.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1;

   c1.push_back( 10 );
   if ( c1.empty( ) )
      cout << "The deque is empty." << endl;
   else
      cout << "The deque is not empty." << endl;
}
```

Output

```
The deque is not empty.
```

deque::end

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

```
const_iterator end() const;

iterator end();
```

Return Value

A random-access iterator that addresses the location succeeding the last element in a deque. If the deque is empty, then deque::end == deque::begin.

Remarks

end is used to test whether an iterator has reached the end of its deque.

Example

```
// deque_end.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1;
   deque <int>::iterator c1_Iter;

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

   c1_Iter = c1.end( );
   c1_Iter--;
   cout << "The last integer of c1 is " << *c1_Iter << endl;

   c1_Iter--;
 *c1_Iter = 400;
   cout << "The new next-to-last integer of c1 is " << *c1_Iter << endl;

   // If a const iterator had been declared instead with the line:
   // deque <int>::const_iterator c1_Iter;
   // an error would have resulted when inserting the 400

   cout << "The deque is now:";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
}
```

Output

```
The last integer of c1 is 30
The new next-to-last integer of c1 is 400
The deque is now: 10 400 30
```

deque::erase

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

```
iterator erase(iterator _Where);

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

Parameters

`_Where`
Position of the element to be removed from the deque.

`first`
Position of the first element removed from the deque.

`last`
Position just beyond the last element removed from the deque.

Return Value

A random-access iterator that designates the first element remaining beyond any elements removed, or a pointer to the end of the deque if no such element exists.

Remarks

For more information on **erase**, see deque::erase and deque::clear.

erase never throws an exception.

Example

```
// deque_erase.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1;
   deque <int>::iterator Iter;

   c1.push_back( 10 );
   c1.push_back( 20 );
   c1.push_back( 30 );
   c1.push_back( 40 );
   c1.push_back( 50 );
   cout << "The initial deque is: ";
   for ( Iter = c1.begin( ); Iter != c1.end( ); Iter++ )
      cout << *Iter << " ";
   cout << endl;
   c1.erase( c1.begin( ) );
   cout << "After erasing the first element, the deque becomes:  ";
```

```
   for ( Iter = c1.begin( ); Iter != c1.end( ); Iter++ )
      cout << *Iter << " ";
   cout << endl;
   Iter = c1.begin( );
   Iter++;
   c1.erase( Iter, c1.end( ) );
   cout << "After erasing all elements but the first, deque becomes: ";
   for ( Iter = c1.begin( ); Iter != c1.end( ); Iter++ )
      cout << *Iter << " ";
   cout << endl;
}
```

Output

```
The initial deque is: 10 20 30 40 50
After erasing the first element, the deque becomes:  20 30 40 50
After erasing all elements but the first, deque becomes: 20
```

deque::front

Returns a reference to the first element in a deque.

```
reference front();

const_reference front() const;
```

Return Value

If the deque is empty, the return is undefined.

Remarks

If the return value of `front` is assigned to a `const_reference`, the deque object cannot be modified. If the return value of `front` is assigned to a **reference**, the deque 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 deque. See Checked Iterators for more information.

Example

```
// deque_front.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>
```

```
int main( )
{
   using namespace std;
   deque <int> c1;

   c1.push_back( 10 );
   c1.push_back( 11 );

   int& i = c1.front( );
   const int& ii = c1.front( );

   cout << "The first integer of c1 is " << i << endl;
   i++;
   cout << "The second integer of c1 is " << ii << endl;
}
```

Output

```
The first integer of c1 is 10
The second integer of c1 is 11
```

deque::get_allocator

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

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

Return Value

The allocator used by the deque.

Remarks

Allocators for the deque 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

```
// deque_get_allocator.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>
```

```
int main( )
{
   using namespace std;
   // The following lines declare objects that use the default allocator.
   deque <int> c1;
   deque <int, allocator<int> > c2 = deque <int, allocator<int> >( allocator<int>(
) );

   // c3 will use the same allocator class as c1
   deque <int> c3( c1.get_allocator( ) );

   deque <int>::allocator_type xlst = c1.get_allocator( );
   // You can now call functions on the allocator class used by c1
}
```

deque::insert

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

```
iterator insert(
    const_iterator Where,
    const Type& Val);

iterator insert(
    const_iterator Where,
    Type&& Val);

void insert(
    iterator Where,
    size_type Count,
    const Type& Val);

template <class InputIterator>
void insert(
    iterator Where,
    InputIterator First,
    InputIterator Last);

iterator insert(
    iterator Where,initializer_list<Type>
IList);
```

Parameters

Parameter	Description
`Where`	The position in the target deque where the first element is inserted.
`Val`	The value of the element being inserted into the deque.
`Count`	The number of elements being inserted into the deque.
`First`	The position of the first element in the range of elements in the argument deque to be copied.
`Last`	The position of the first element beyond the range of elements in the argument deque to be copied.
`IList`	The initializer_list of elements to insert.

Return Value

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

Remarks

Any insertion operation can be expensive.

deque::iterator

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

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

Remarks

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

Example

See the example for begin.

deque::max_size

Returns the maximum length of the deque.

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

Return Value

The maximum possible length of the deque.

Example

```
// deque_max_size.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1;
   deque <int>::size_type i;

   i = c1.max_size( );
   cout << "The maximum possible length of the deque is " << i << "." << endl;
}
```

deque::operator[]

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

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

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

Parameters

_Pos
The position of the deque element to be referenced.

Return Value

A reference to the element whose position is specified in the argument. If the position specified is greater than the size of the deque, the result is undefined.

Remarks

If the return value of `operator[]` is assigned to a `const_reference`, the deque object cannot be modified. If the return value of `operator[]` is assigned to a **reference**, the deque 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 outside the bounds of the deque. See Checked Iterators for more information.

Example

```
// deque_op_ref.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1;

   c1.push_back( 10 );
   c1.push_back( 20 );
   cout << "The first integer of c1 is " << c1[0] << endl;
   int& i = c1[1];
   cout << "The second integer of c1 is " << i << endl;

}
```

Output

```
The first integer of c1 is 10
The second integer of c1 is 20
```

deque::operator=

Replaces the elements of this deque using the elements from another deque.

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

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

Parameters

Parameter	Description
`right`	The deque that provides the new content.

Remarks

The first override copies elements to this deque from `right`, the source of the assignment. The second override moves elements to this deque from `right`.

Elements that are contained in this deque before the operator executes are removed.

Example

```
// deque_operator_as.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>
using namespace std;

typedef deque<int> MyDeque;

template<typename MyDeque> struct S;

template<typename MyDeque> struct S<MyDeque&> {
  static void show( MyDeque& d ) {
    MyDeque::const_iterator iter;
    for (iter = d.cbegin(); iter != d.cend(); iter++)
       cout << *iter << " ";
    cout << endl;
  }
};

template<typename MyDeque> struct S<MyDeque&&> {
  static void show( MyDeque&& d ) {
    MyDeque::const_iterator iter;
    for (iter = d.cbegin(); iter != d.cend(); iter++)
       cout << *iter << " ";
cout << " via unnamed rvalue reference " << endl;
  }
};

int main( )
{
   MyDeque d1, d2;

   d1.push_back(10);
   d1.push_back(20);
   d1.push_back(30);
   d1.push_back(40);
   d1.push_back(50);
```

```
    cout << "d1 = " ;
    S<MyDeque&>::show( d1 );

    d2 = d1;
    cout << "d2 = ";
    S<MyDeque&>::show( d2 );

    cout << "     ";
    S<MyDeque&&>::show ( move< MyDeque& > (d1) );
  }
```

deque::pointer

Provides a pointer to an element in a deque.

unstlib

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

Remarks

A type **pointer** can be used to modify the value of an element. An iterator is more commonly used to access a deque element.

deque::pop_back

Deletes the element at the end of the deque.

```
void pop_back();
```

Remarks

The last element must not be empty. pop_back never throws an exception.

Example

```
// deque_pop_back.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
```

```
{
   using namespace std;
   deque <int> c1;

   c1.push_back( 1 );
   c1.push_back( 2 );
   cout << "The first element is: " << c1.front( ) << endl;
   cout << "The last element is: " << c1.back( ) << endl;

   c1.pop_back( );
   cout << "After deleting the element at the end of the deque, the "
      "last element is: " << c1.back( ) << endl;
}
```

Output

```
The first element is: 1
The last element is: 2
After deleting the element at the end of the deque, the last element is: 1
```

deque::pop_front

Deletes the element at the beginning of the deque.

```
void pop_front();
```

Remarks

The first element must not be empty. pop_front never throws an exception.

Example

```
// deque_pop_front.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1;

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

```
   cout << "The first element is: " << c1.front( ) << endl;
   cout << "The second element is: " << c1.back( ) << endl;

   c1.pop_front( );
   cout << "After deleting the element at the beginning of the "
      "deque, the first element is: " << c1.front( ) << endl;
}
```

Output

```
The first element is: 1
The second element is: 2
After deleting the element at the beginning of the deque, the first element is: 2
```

deque::push_back

Adds an element to the end of the deque.

```
void push_back(const Type& val);

void push_back(Type&& val);
```

Parameters

Parameter	Description
`val`	The element added to the end of the deque.

Remarks

If an exception is thrown, the deque is left unaltered and the exception is rethrown.

deque::push_front

Adds an element to the beginning of the deque.

```
    void push_front(const Type& val);
```

```
void push_front(Type&& val);
```

Parameters

Parameter	Description
`val`	The element added to the beginning of the deque.

Remarks

If an exception is thrown, the deque is left unaltered and the exception is rethrown.

Example

```
// deque_push_front.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>
#include <string>

int main( )
{
   using namespace std;
   deque <int> c1;

   c1.push_front( 1 );
   if ( c1.size( ) != 0 )
      cout << "First element: " << c1.front( ) << endl;

   c1.push_front( 2 );
   if ( c1.size( ) != 0 )
      cout << "New first element: " << c1.front( ) << endl;

// move initialize a deque of strings
   deque <string> c2;
   string str("a");

   c2.push_front( move( str ) );
   cout << "Moved first element: " << c2.front( ) << endl;
}
```

Output

```
First element: 1
New first element: 2
```

```
Moved first element: a
```

deque::rbegin

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

```
const_reverse_iterator rbegin() const;

reverse_iterator rbegin();
```

Return Value

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

Remarks

`rbegin` is used with a reversed deque just as begin is used with a deque.

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

`rbegin` can be used to iterate through a deque backwards.

Example

```
// deque_rbegin.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1;
   deque <int>::iterator c1_Iter;
   deque <int>::reverse_iterator c1_rIter;

   // If the following line had replaced the line above, an error
   // would have resulted in the line modifying an element
   // (commented below) because the iterator would have been const
   // deque <int>::const_reverse_iterator c1_rIter;

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

```
   c1_rIter = c1.rbegin( );
   cout << "Last element in the deque is " << *c1_rIter << "." << endl;

   cout << "The deque contains the elements: ";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << *c1_Iter << " ";
   cout << "in that order.";
   cout << endl;

   // rbegin can be used to iterate through a deque in reverse order
   cout << "The reversed deque is: ";
   for ( c1_rIter = c1.rbegin( ); c1_rIter != c1.rend( ); c1_rIter++ )
      cout << *c1_rIter << " ";
   cout << endl;

   c1_rIter = c1.rbegin( );
 *c1_rIter = 40;  // This would have caused an error if a
                  // const_reverse iterator had been declared as
                  // noted above
   cout << "Last element in deque is now " << *c1_rIter << "." << endl;
}
```

Output

```
Last element in the deque is 30.
The deque contains the elements: 10 20 30 in that order.
The reversed deque is: 30 20 10
Last element in deque is now 40.
```

deque::reference

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

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

Example

```
// deque_reference.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
```

```
{
   using namespace std;
   deque <int> c1;

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

   const int &i = c1.front( );
   int &j = c1.back( );
   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
```

deque::rend

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

```
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 deque (the location that had preceded the first element in the unreversed deque).

Remarks

`rend` is used with a reversed deque just as end is used with a deque.

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

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

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

Example

```
// deque_rend.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;

   deque <int> c1;
   deque <int>::iterator c1_Iter;
   deque <int>::reverse_iterator c1_rIter;
   // If the following line had replaced the line above, an error
   // would have resulted in the line modifying an element
   // (commented below) because the iterator would have been const
   // deque <int>::const_reverse_iterator c1_rIter;

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

   c1_rIter = c1.rend( );
   c1_rIter --; // Decrementing a reverse iterator moves it forward
                // in the deque (to point to the first element here)
   cout << "The first element in the deque is: " << *c1_rIter << endl;

   cout << "The deque is: ";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << *c1_Iter << " ";
   cout << endl;

   // rend can be used to test if an iteration is through all of
   // the elements of a reversed deque
   cout << "The reversed deque is: ";
   for ( c1_rIter = c1.rbegin( ); c1_rIter != c1.rend( ); c1_rIter++ )
      cout << *c1_rIter << " ";
   cout << endl;

   c1_rIter = c1.rend( );
   c1_rIter--; // Decrementing the reverse iterator moves it backward
               // in the reversed deque (to the last element here)
 *c1_rIter = 40; // This modification of the last element would
                 // have caused an error if a const_reverse
                 // iterator had been declared (as noted above)
   cout << "The modified reversed deque is: ";
   for ( c1_rIter = c1.rbegin( ); c1_rIter != c1.rend( ); c1_rIter++ )
      cout << *c1_rIter << " ";
   cout << endl;
}
```

Output

```
The first element in the deque is: 10
The deque is: 10 20 30
The reversed deque is: 30 20 10
The modified reversed deque is: 30 20 40
```

deque::resize

Specifies a new size for a deque.

```
void resize(size_type _Newsize);

void resize(size_type _Newsize, Type val);
```

Parameters

`_Newsize`
The new size of the deque.

`val`
The value of the new elements to be added to the deque if the new size is larger that the original size. If the value is omitted, the new elements are assigned the default value for the class.

Remarks

If the deque's size is less than the requested size, `_Newsize`, elements are added to the deque until it reaches the requested size.

If the deque's size is larger than the requested size, the elements closest to the end of the deque are deleted until the deque reaches the size `_Newsize`.

If the present size of the deque is the same as the requested size, no action is taken.

size reflects the current size of the deque.

Example

```
// deque_resize.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1;
```

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

    c1.resize( 4,40 );
    cout << "The size of c1 is: " << c1.size( ) << endl;
    cout << "The value of the last element is " << c1.back( ) << endl;

    c1.resize( 5 );
    cout << "The size of c1 is now: " << c1.size( ) << endl;
    cout << "The value of the last element is now " << c1.back( ) << endl;

    c1.resize( 2 );
    cout << "The reduced size of c1 is: " << c1.size( ) << endl;
    cout << "The value of the last element is now " << c1.back( ) << endl;
}
```

Output

```
The size of c1 is: 4
The value of the last element is 40
The size of c1 is now: 5
The value of the last element is now 0
The reduced size of c1 is: 2
The value of the last element is now 20
```

deque::reverse_iterator

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

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

Remarks

A type `reverse_iterator` is use to iterate through the deque.

Example

See the example for rbegin.

deque::shrink_to_fit

Discards excess capacity.

```
void shrink_to_fit();
```

Remarks

There is no portable way to determine if `shrink_to_fit` reduces the storage used by a deque.

Example

```
// deque_shrink_to_fit.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

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

   v1.push_back( 1 );
   v1.push_back( 2 );
   cout << "Current size of v1 = "
      << v1.size( ) << endl;
   v1.shrink_to_fit();
   cout << "Current size of v1 = "
      << v1.size( ) << endl;
}
```

Output

```
Current size of v1 = 1
Current size of v1 = 1
```

deque::size

Returns the number of elements in the deque.

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

Return Value

The current length of the deque.

Example

```
// deque_size.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1;
   deque <int>::size_type i;

   c1.push_back( 1 );
   i = c1.size( );
   cout << "The deque length is " << i << "." << endl;

   c1.push_back( 2 );
   i = c1.size( );
   cout << "The deque length is now " << i << "." << endl;
}
```

Output

```
The deque length is 1.
The deque length is now 2.
```

deque::size_type

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

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

Example

See the example for size.

deque::swap

Exchanges the elements of two deques.

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

friend void swap(deque<Type, Allocator>& left, deque<Type, Allocator>& right)
template <class Type, class Allocator>
void swap(deque<Type, Allocator>& left, deque<Type, Allocator>& right);
```

Parameters

`right`
The deque providing the elements to be swapped, or the deque whose elements are to be exchanged with those of the deque `left`.

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

Example

```
// deque_swap.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>

int main( )
{
   using namespace std;
   deque <int> c1, c2, c3;
   deque <int>::iterator c1_Iter;

   c1.push_back( 1 );
   c1.push_back( 2 );
   c1.push_back( 3 );
   c2.push_back( 10 );
   c2.push_back( 20 );
   c3.push_back( 100 );

   cout << "The original deque c1 is:";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;

   c1.swap( c2 );

   cout << "After swapping with c2, deque c1 is:";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;
```

```
   swap( c1,c3 );

   cout << "After swapping with c3, deque c1 is:";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;

   swap<>(c1, c2);
   cout << "After swapping with c2, deque c1 is:";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;
}
```

Output

```
The original deque c1 is: 1 2 3
After swapping with c2, deque c1 is: 10 20
After swapping with c3, deque c1 is: 100
After swapping with c2, deque c1 is: 1 2 3
```

deque::value_type

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

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

Remarks

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

Example

```
// deque_value_type.cpp
// compile with: /EHsc
#include <deque>
#include <iostream>
int main( )
{
   using namespace std;
   deque<int>::value_type AnInt;
   AnInt = 44;
   cout << AnInt << endl;
```

```
}
```

Output

```
44
```

See Also

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