list 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 list Class.

The STL list class is a template class of sequence containers that maintain their elements in a linear arrangement and allow efficient insertions and deletions at any location within the sequence. The sequence is stored as a bidirectional linked list of elements, each containing a member of some type *Type*.

Syntax

C++

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

Parameters
Type
The element data type to be stored in the list.

`Allocator`
The type that represents the stored allocator object that encapsulates details about the list'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 class deque container is superior when random access is needed and insertions and deletions at both the beginning and the end of a sequence are at a premium.

The list member functions merge, reverse, unique, remove, and remove_if have been optimized for operation on list objects and offer a high-performance alternative to their generic counterparts.

List reallocation occurs when a member function must insert or erase elements of the list. In all such cases, only iterators or references that point at erased portions of the controlled sequence become invalid.

Include the STL standard header <list> to define the container template class list and several supporting templates.

Constructors

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

Typedefs

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

Member Functions

assign	Erases elements from a list and copies a new set of elements to the target list.
back	Returns a reference to the last element of a list.

begin	Returns an iterator addressing the first element in a list.
list::cbegin	Returns a const iterator addressing the first element in a list.
list::cend	Returns a const iterator that addresses the location succeeding the last element in a list.
list::clear	Erases all the elements of a list.
list::crbegin	Returns a const iterator addressing the first element in a reversed list.
list::crend	Returns a const iterator that addresses the location succeeding the last element in a reversed list.
list::emplace	Inserts an element constructed in place into a list at a specified position.
list::emplace_back	Adds an element constructed in place to the end of a list.
list::emplace_front	Adds an element constructed in place to the beginning of a list.
empty	Tests if a list is empty.
end	Returns an iterator that addresses the location succeeding the last element in a list.
erase	Removes an element or a range of elements in a list from specified positions.
front	Returns a reference to the first element in a list.
get_allocator	Returns a copy of the `allocator` object used to construct a list.
insert	Inserts an element or a number of elements or a range of elements into a list at a specified position.
max_size	Returns the maximum length of a list.
merge	Removes the elements from the argument list, inserts them into the target list, and orders the new, combined set of elements in ascending order or in some other specified order.
pop_back	Deletes the element at the end of a list.
pop_front	Deletes the element at the beginning of a list.
push_back	Adds an element to the end of a list.
push_front	Adds an element to the beginning of a list.
rbegin	Returns an iterator addressing the first element in a reversed list.
remove	Erases elements in a list that match a specified value.

remove_if	Erases elements from the list for which a specified predicate is satisfied.
rend	Returns an iterator that addresses the location succeeding the last element in a reversed list.
resize	Specifies a new size for a list.
reverse	Reverses the order in which the elements occur in a list.
size	Returns the number of elements in a list.
sort	Arranges the elements of a list in ascending order or with respect to some other order relation.
splice	Removes elements from the argument list and inserts them into the target list.
swap	Exchanges the elements of two lists.
unique	Removes adjacent duplicate elements or adjacent elements that satisfy some other binary predicate from the list.

Operators

list::operator=	Replaces the elements of the list with a copy of another list.

Requirements

Header: <list>

list::allocator_type

A type that represents the allocator class for a list object.

```
typedef Allocator allocator_type;
```

Remarks

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

Example

See the example for get_allocator.

list::assign

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

```
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 from the argument list.

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

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

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

`IList`
The initializer_list that contains the elements to be inserted.

Remarks

After erasing any existing elements in the target list, assign either inserts a specified range of elements from the original list or from some other list into the target list or inserts copies of a new element of a specified value into the target list

Example

```
// list_assign.cpp
// compile with: /EHsc
#include <list>
```

```
#include <iostream>

int main()
{
    using namespace std;
    list<int> c1, c2;
    list<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);

    cout << "c1 =";
    for (auto c : c1)
        cout << " " << c;
    cout << endl;

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

    c1.assign(7, 4);
    cout << "c1 =";
    for (auto c : c1)
        cout << " " << c;
    cout << endl;

    c1.assign({ 10, 20, 30, 40 });
    cout << "c1 =";
    for (auto c : c1)
        cout << " " << c;
    cout << endl;
}
```

Output

```
c1 = 10 20 30c1 = 50 60c1 = 4 4 4 4 4 4 4c1 = 10 20 30 40
```

list::back

Returns a reference to the last element of a list.

```
reference back();

const_reference back() const;
```

Return Value

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

Remarks

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

Example

```
// list_back.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <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
```

list::begin

Returns an iterator addressing the first element in a list.

```
const_iterator begin() const;

iterator begin();
```

Return Value

A bidirectional iterator addressing the first element in the list or to the location succeeding an empty list.

Remarks

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

Example

```
// list_begin.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

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

list::cbegin

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

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

Return Value

A `const` bidirectional-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++

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

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

list::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` bidirectional-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.

list::clear

Erases all the elements of a list.

```
void clear();
```

Example

```
// list_clear.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

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

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

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

```
    }
```

Output

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

list::const_iterator

A type that provides a bidirectional iterator that can read a **const** element in a list.

```
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.

list::const_pointer

Provides a pointer to a `const` element in a list.

unstlib

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

Remarks

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

In most cases, an iterator should be used to access the elements in a list object.

list::const_reference

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

```
// list_const_ref.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

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

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

   const list <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 because c2 is const
   // c2.push_back( 30 );
}
```

Output

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

list::const_reverse_iterator

A type that provides a bidirectional iterator that can read any **const** element in a list.

```
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 list in reverse.

Example

See the example for rbegin.

list::crbegin

Returns a const iterator addressing the first element in a reversed list.

```
const_reverse_iterator rbegin() const;
```

Return Value

A const reverse bidirectional iterator addressing the first element in a reversed list (or addressing what had been the last element in the unreversed `list`).

Remarks

`crbegin` is used with a reversed list just as list::begin is used with a `list`.

With the return value of `crbegin`, the list object cannot be modified. list::rbegin can be used to iterate through a list backwards.

Example

```
// list_crbegin.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;
   list <int>::const_reverse_iterator c1_crIter;

   c1.push_back( 10 );
   c1.push_back( 20 );
   c1.push_back( 30 );
   c1_crIter = c1.crbegin( );
   cout << "The last element in the list is " << *c1_crIter << "." << endl;
}
```

Output

```
The last element in the list is 30.
```

list::crend

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

```
const_reverse_iterator rend() const;
```

Return Value

A const reverse bidirectional iterator that addresses the location succeeding the last element in a reversed list (the location that had preceded the first element in the unreversed `list`).

Remarks

`crend` is used with a reversed list just as list::end is used with a `list`.

With the return value of `crend`, the `list` object cannot be modified.

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

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

Example

```
// list_crend.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;
   list <int>::const_reverse_iterator c1_crIter;

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

   c1_crIter = c1.crend( );
   c1_crIter --;  // Decrementing a reverse iterator moves it forward in
                 // the list (to point to the first element here)
   cout << "The first element in the list is: " << *c1_crIter << endl;
}
```

Output

```
The first element in the list is: 10
```

list::difference_type

A signed integer type that can be used to represent the number of elements of a list in a range between elements pointed to by iterators.

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

Remarks

The `difference_type` is the type returned when subtracting or incrementing through iterators of the container. The `difference_type` is typically used to represent the number of elements in the range [`first`, `last`) between the iterators `first` and `last`, includes the element pointed to by `first` and the range of elements up to, but not including, the element pointed to by `last`.

Note that although `difference_type` is available for all iterators that satisfy the requirements of an input iterator, which includes the class of bidirectional iterators supported by reversible containers like set, subtraction between iterators is only supported by random-access iterators provided by a random-access container, such as vector Class.

Example

```
// list_diff_type.cpp
// compile with: /EHsc
#include <iostream>
#include <list>
#include <algorithm>

int main( )
{
   using namespace std;

   list <int> c1;
   list <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( );

     list <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.
```

list::emplace

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

```
void emplace_back(iterator _Where, Type&& val);
```

Parameters

Parameter	Description
`_Where`	The position in the target list where the first element is inserted.
`val`	The element added to the end of the `list`.

Remarks

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

Example

```
// list_emplace.cpp
// compile with: /EHsc
#include <list>
#include <iostream>
#include <string>

int main( )
{
   using namespace std;
   list <string> c2;
   string str("a");

   c2.emplace(c2.begin(), move( str ) );
   cout << "Moved first element: " << c2.back( ) << endl;
}
```

Output

```
Moved first element: a
```

list::emplace_back

Adds an element constructed in place to the beginning of a list.

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

Parameters

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

Remarks

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

Example

```
// list_emplace_back.cpp
// compile with: /EHsc
#include <list>
#include <iostream>
#include <string>

int main( )
{
   using namespace std;
   list <string> c2;
   string str("a");

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

Output

```
Moved first element: a
```

list::emplace_front

Adds an element constructed in place to the beginning of a list.

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

Parameters

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

Remarks

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

Example

```
// list_emplace_front.cpp
// compile with: /EHsc
#include <list>
#include <iostream>
#include <string>

int main( )
{
   using namespace std;
   list <string> c2;
   string str("a");

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

Output

```
Moved first element: a
```

list::empty

Tests if a list is empty.

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

Return Value

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

Example

```
// list_empty.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

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

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

Output

```
The list is not empty.
```

list::end

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

```
const_iterator end() const;

iterator end();
```

Return Value

A bidirectional iterator that addresses the location succeeding the last element in a list. If the list is empty, then `list::end == list::begin`.

Remarks

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

Example

```
// list_end.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;
   list <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:
   // list <int>::const_iterator c1_Iter;
   // an error would have resulted when inserting the 400

   cout << "The list 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 list is now: 10 400 30
```

list::erase

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

```
iterator erase(iterator _Where);

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

Parameters

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

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

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

Return Value

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

Remarks

No reallocation occurs, so iterators and references become invalid only for the erased elements.

erase never throws an exception.

Example

```
// list_erase.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;
   list <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 list is:";
   for ( Iter = c1.begin( ); Iter != c1.end( ); Iter++ )
      cout << " " << *Iter;
   cout << endl;

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

Output

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

list::front

Returns a reference to the first element in a list.

```
reference front();

const_reference front() const;
```

Return Value

If the list is empty, the return is undefined.

Remarks

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

Example

```
// list_front.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

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

   c1.push_back( 10 );

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

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

Output

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

list::get_allocator

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

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

Return Value

The allocator used by the list.

Remarks

Allocators for the list 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

```
// list_get_allocator.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

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

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

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

list::insert

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

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

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

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

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

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

Parameters

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

Example

```cpp
// list_class_insert.cpp
// compile with: /EHsc
#include <list>
#include <iostream>
#include <string>

int main()
{
    using namespace std;
    list <int> c1, c2;
    list <int>::iterator Iter;

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

    cout << "c1 =";
    for (auto c : c1)
        cout << " " << c;
    cout << endl;

    Iter = c1.begin();
    Iter++;
    c1.insert(Iter, 100);
    cout << "c1 =";
    for (auto c : c1)
        cout << " " << c;
    cout << endl;

    Iter = c1.begin();
    Iter++;
    Iter++;
    c1.insert(Iter, 2, 200);

    cout << "c1 =";
    for(auto c : c1)
        cout << " " << c;
    cout << endl;

    c1.insert(++c1.begin(), c2.begin(), --c2.end());

    cout << "c1 =";
    for (auto c : c1)
        cout << " " << c;
    cout << endl;

    // initialize a list of strings by moving
    list < string > c3;
    string str("a");

    c3.insert(c3.begin(), move(str));
    cout << "Moved first element: " << c3.front() << endl;

    // Assign with an initializer_list
    list <int> c4{ {1, 2, 3, 4} };
    c4.insert(c4.begin(), { 5, 6, 7, 8 });

    cout << "c4 =";
    for (auto c : c4)
        cout << " " << c;
    cout << endl;
}
```

list::iterator

A type that provides a bidirectional iterator that can read or modify any element in a list.

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

Remarks

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

Example

See the example for begin.

list::list

Constructs a list 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 list.

```
list();

explicit list(
    const Allocator& Al);

explicit list(
    size_type Count);

list(
    size_type Count,
    const Type& Val);

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

list(
    const list& Right);

list(
    list&& Right);

list(
    initializer_list<Type> IList,
    const Allocator& Al);

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

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

Parameters

Parameter	Description
`Al`	The allocator class to use with this object.
`Count`	The number of elements in the list constructed.
`Val`	The value of the elements in the list.
`Right`	The list of which the constructed list is to be a copy.
`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.
`IList`	The initializer_list that contains the elements to be copied.

Remarks

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

get_allocator returns a copy of the allocator object used to construct a list.

The first two constructors specify an empty initial list, the second specifying 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 list `Right`.

The seventh constructor moves the list `Right`.

The eighth constructor uses an initializer_list to specify the elements.

The next two constructors copy the range `[First, Last)` of a list.

None of the constructors perform any interim reallocations.

Example

C++

```cpp
// list_class_list.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main()
{
    using namespace std;
    // Create an empty list c0
    list <int> c0;

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

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

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

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

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

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

    cout << "c1 =";
    for (auto c : c1)
        cout << " " << c;
    cout << endl;

    cout << "c2 =";
    for (auto c : c2)
        cout << " " << c;
    cout << endl;

    cout << "c3 =";
    for (auto c : c3)
        cout << " " << c;
    cout << endl;

    cout << "c4 =";
    for (auto c : c4)
        cout << " " << c;
    cout << endl;

    cout << "c5 =";
    for (auto c : c5)
        cout << " " << c;
    cout << endl;

    cout << "c6 =";
    for (auto c : c6)
        cout << " " << c;
    cout << endl;

    // Move list c6 to list c7
    list <int> c7(move(c6));
    cout << "c7 =";
    for (auto c : c7)
        cout << " " << c;
    cout << endl;

    // Construct with initializer_list
    list<int> c8({ 1, 2, 3, 4 });
    cout << "c8 =";
    for (auto c : c8)
        cout << " " << c;
    cout << endl;
```

```
}
```

Output

```
c1 = 0 0 0c2 = 2 2 2 2 2c3 = 1 1 1c4 = 2 2 2 2 2c5 = 2 2c6 = 2 2 2c7 = 2 2 2c8 = 1
2 3 4
```

list::max_size

Returns the maximum length of a list.

unstlib

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

Return Value

The maximum possible length of the list.

Example

```
// list_max_size.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

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

   i = c1.max_size( );
   cout << "Maximum possible length of the list is " << i << "." << endl;
}
```

list::merge

Removes the elements from the argument list, inserts them into the target list, and orders the new, combined set of elements in ascending order or in some other specified order.

```
void merge(
    list<Type, Allocator>& right);

template <class Traits>
void merge(
    list<Type, Allocator>& right,
    Traits comp);
```

Parameters

`right`
The argument list to be merged with the target list.

`comp`
The comparison operator used to order the elements of the target list.

Remarks

The argument list `right` is merged with the target list.

Both argument and target lists must be ordered with the same comparison relation by which the resulting sequence is to be ordered. The default order for the first member function is ascending order. The second member function imposes the user-specified comparison operation `comp` of class **Traits**.

Example

```
// list_merge.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1, c2, c3;
   list <int>::iterator c1_Iter, c2_Iter, c3_Iter;

   c1.push_back( 3 );
   c1.push_back( 6 );
   c2.push_back( 2 );
   c2.push_back( 4 );
   c3.push_back( 5 );
   c3.push_back( 1 );

   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;

   c2.merge( c1 );  // Merge c1 into c2 in (default) ascending order
   c2.sort( greater<int>( ) );
   cout << "After merging c1 with c2 and sorting with >: 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;

   c2.merge( c3, greater<int>( ) );
   cout << "After merging c3 with c2 according to the '>' comparison relation: c2
=";
   for ( c2_Iter = c2.begin( ); c2_Iter != c2.end( ); c2_Iter++ )
      cout << " " << *c2_Iter;
   cout << endl;
}
```

Output

```
c1 = 3 6
c2 = 2 4
After merging c1 with c2 and sorting with >: c2 = 6 4 3 2
c3 = 5 1
After merging c3 with c2 according to the '>' comparison relation: c2 = 6 5 4 3 2
1
```

list::operator=

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

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

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

Parameters

Parameter	Description

right	The list being copied into the list.

Remarks

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

Example

```
// list_operator_as.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list<int> v1, v2, v3;
   list<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 = forward< list<int> >(v1);
   cout << "v2 = ";
   for (iter = v2.begin(); iter != v2.end(); iter++)
      cout << *iter << " ";
   cout << endl;
}
```

list::pointer

Provides a pointer to an element in a list.

unstlib

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

Remarks

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

In most cases, an iterator should be used to access the elements in a list object.

list::pop_back

Deletes the element at the end of a list.

unstlib

```
void pop_back();
```

Remarks

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

Example

```
// list_pop_back.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <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 list, "
           "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 list, the last element is: 1
```

list::pop_front

Deletes the element at the beginning of a list.

unstlib

```
void pop_front();
```

Remarks

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

Example

```
// list_pop_front.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <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 list, "
         "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 list, the first element is: 2
```

list::push_back

Adds an element to the end of a list.

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

Parameters

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

Remarks

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

Example

```
// list_push_back.cpp
// compile with: /EHsc
#include <list>
#include <iostream>
#include <string>

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

   c1.push_back( 1 );
   if ( c1.size( ) != 0 )
      cout << "Last element: " << c1.back( ) << endl;

   c1.push_back( 2 );
   if ( c1.size( ) != 0 )
      cout << "New last element: " << c1.back( ) << endl;

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

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

Output

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

list::push_front

Adds an element to the beginning of a list.

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

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

Parameters

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

Remarks

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

Example

```
// list_push_front.cpp
// compile with: /EHsc
#include <list>
#include <iostream>
#include <string>

int main( )
{
   using namespace std;
   list <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 list of strings
      list <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
```

list::rbegin

Returns an iterator that addresses the first element in a reversed list.

unstlib

```
const_reverse_iterator rbegin() const;

reverse_iterator rbegin();
```

Return Value

A reverse bidirectional iterator addressing the first element in a reversed list (or addressing what had been the last element in the unreversed list).

Remarks

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

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

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

Example

```
// list_rbegin.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

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

   // If the following line replaced the line above, *c1_rIter = 40;
   // (below) would be an error
   //list <int>::const_reverse_iterator c1_rIter;

   c1.push_back( 10 );
   c1.push_back( 20 );
   c1.push_back( 30 );
   c1_rIter = c1.rbegin( );
   cout << "The last element in the list is " << *c1_rIter << "." << endl;

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

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

   c1_rIter = c1.rbegin( );
 *c1_rIter = 40;
   cout << "The last element in the list is now " << *c1_rIter << "." << endl;
}
```

Output

```
The last element in the list is 30.
The list is: 10 20 30
The reversed list is: 30 20 10
The last element in the list is now 40.
```

list::reference

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

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

Example

```
// list_ref.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

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

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

   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
```

list::remove

Erases elements in a list that match a specified value.

unstlib

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

Parameters

`val`
The value which, if held by an element, will result in that element's removal from the list.

Remarks

The order of the elements remaining is not affected.

Example

```
// list_remove.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;
   list <int>::iterator c1_Iter, c2_Iter;

   c1.push_back( 5 );
   c1.push_back( 100 );
   c1.push_back( 5 );
   c1.push_back( 200 );
   c1.push_back( 5 );
   c1.push_back( 300 );

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

   list <int> c2 = c1;
   c2.remove( 5 );
   cout << "After removing elements with value 5, the list becomes c2 =";
   for ( c2_Iter = c2.begin( ); c2_Iter != c2.end( ); c2_Iter++ )
      cout << " " << *c2_Iter;
   cout << endl;
}
```

Output

```
The initial list is c1 = 5 100 5 200 5 300
After removing elements with value 5, the list becomes c2 = 100 200 300
```

list::remove_if

Erases elements from a list for which a specified predicate is satisfied.

unstlib

```
template <class Predicate>
void remove_if(Predicate pred)
```

Parameters

`pred`
The unary predicate which, if satisfied by an element, results in the deletion of that element from the list.

Example

```
// list_remove_if.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

template <class T> class is_odd : public std::unary_function<T, bool>
{
public:
   bool operator( ) ( T& val )
   {
   return ( val % 2 ) == 1;
   }
};

int main( )
{
   using namespace std;
   list <int> c1;
   list <int>::iterator c1_Iter, c2_Iter;

   c1.push_back( 3 );
   c1.push_back( 4 );
   c1.push_back( 5 );
   c1.push_back( 6 );
   c1.push_back( 7 );
   c1.push_back( 8 );

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

   list <int> c2 = c1;
   c2.remove_if( is_odd<int>( ) );

   cout << "After removing the odd elements, "
        << "the list becomes c2 =";
   for ( c2_Iter = c2.begin( ); c2_Iter != c2.end( ); c2_Iter++ )
      cout << " " << *c2_Iter;
   cout << endl;
}
```

Output

```
The initial list is c1 = 3 4 5 6 7 8
After removing the odd elements, the list becomes c2 = 4 6 8
```

list::rend

Returns an iterator that addresses the location that follows the last element in a reversed list.

unstlib

```
const_reverse_iterator rend() const;

reverse_iterator rend();
```

Return Value

A reverse bidirectional iterator that addresses the location succeeding the last element in a reversed list (the location that had preceded the first element in the unreversed list).

Remarks

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

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

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

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

Example

```
// list_rend.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1;
   list <int>::iterator c1_Iter;
   list <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
   // list <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 list (to point to the first element here)
   cout << "The first element in the list is: " << *c1_rIter << endl;

   cout << "The list 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 list
   cout << "The reversed list 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 list (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 list is:";
   for ( c1_rIter = c1.rbegin( ); c1_rIter != c1.rend( ); c1_rIter++ )
      cout << " " << *c1_rIter;
   cout << endl;
}
```

Output

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

list::resize

Specifies a new size for a list.

unstlib

```
void resize(size_type _Newsize);

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

Parameters

`_Newsize`
The new size of the list.

`val`
The value of the new elements to be added to the list 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 list's size is less than the requested size, `_Newsize`, elements are added to the list until it reaches the requested size.

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

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

size reflects the current size of the list.

Example

```
// list_resize.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <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
```

list::reverse

Reverses the order in which the elements occur in a list.

unstlib

```
void reverse();
```

Example

```
// list_reverse.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

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

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

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

   c1.reverse( );
```

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

Output

```
c1 = 10 20 30
Reversed c1 = 30 20 10
```

list::reverse_iterator

A type that provides a bidirectional iterator that can read or modify an element in a reversed list.

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

Remarks

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

Example

See the example for rbegin.

list::size

Returns the number of elements in a list.

unstlib

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

Return Value

The current length of the list.

Example

```
// list_size.cpp
// compile with: /EHsc
```

```
#include <list>
#include <iostream>

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

   c1.push_back( 5 );
   i = c1.size( );
   cout << "List length is " << i << "." << endl;

   c1.push_back( 7 );
   i = c1.size( );
   cout << "List length is now " << i << "." << endl;
}
```

Output

```
List length is 1.
List length is now 2.
```

list::size_type

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

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

Example

See the example for size.

list::sort

Arranges the elements of a list in ascending order or with respect to some other user-specified order.

unstlib

```
void sort();

template <class Traits>
void sort(Traits comp);
```

Parameters

`comp`
The comparison operator used to order successive elements.

Remarks

The first member function puts the elements in ascending order by default.

The member template function orders the elements according to the user-specified comparison operation `comp` of class **Traits**.

Example

```
// list_sort.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

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

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

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

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

   c1.sort( greater<int>( ) );
   cout << "After sorting with 'greater than' operation, c1 =";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;
}
```

Output

```
Before sorting: c1 = 20 10 30
After sorting c1 = 10 20 30
After sorting with 'greater than' operation, c1 = 30 20 10
```

list::splice

Removes elements from a source list and inserts them into a destination list.

```
// insert the entire source list
void splice(const_iterator Where, list<Type, Allocator>& Source);

void splice(const_iterator Where, list<Type, Allocator>&& Source);


// insert one element of the source list
void splice(const_iterator Where, list<Type, Allocator>& Source, const_iterator
Iter);

void splice(const_iterator Where, list<Type, Allocator>&& Source, const_iterator
Iter);


// insert a range of elements from the source list
void splice(const_iterator Where, list<Type, Allocator>& Source, const_iterator
First, const_iterator Last);

    void splice(const_iterator Where, list<Type, Allocator>&& Source,
const_iterator First, const_iterator Last);
```

Parameters

`Where`
The position in the destination list before which to insert.

`Source`
The source list that is to be inserted into the destination list.

`Iter`
The element to be inserted from the source list.

`First`
The first element in the range to be inserted from the source list.

`Last`
The first position beyond the last element in the range to be inserted from the source list.

Remarks

The first pair of member functions inserts all elements in the source list into the destination list before the position referred to by `Where` and removes all elements from the source list. (`&Source` must not equal `this`.)

The second pair of member functions inserts the element referred to by `Iter` before the position in the destination list referred to by `Where` and removes `Iter` from the source list. (If `Where == Iter || Where == ++Iter`, no change occurs.)

The third pair of member functions inserts the range designated by [`First`, `Last`) before the element in the destination list referred to by `Where` and removes that range of elements from the source list. (If `&Source == this`, the range `[First, Last)` must not include the element pointed to by `Where`.)

If the ranged splice inserts `N` elements, and `&Source != this`, an object of class iterator is incremented `N` times.

In all cases iterators, pointers, or references that refer to spliced elements remain valid and are transferred to the destination container.

Example

C++

```cpp
// list_splice.cpp
// compile with: /EHsc /W4
#include <list>
#include <iostream>

using namespace std;

template <typename S> void print(const S& s) {
    cout << s.size() << " elements: ";

    for (const auto& p : s) {
        cout << "(" << p << ") ";
    }

    cout << endl;
}

int main()
{
    list<int> c1{10,11};
    list<int> c2{20,21,22};
    list<int> c3{30,31};
    list<int> c4{40,41,42,43};

    list<int>::iterator where_iter;
    list<int>::iterator first_iter;
    list<int>::iterator last_iter;

    cout << "Beginning state of lists:" << endl;
    cout << "c1 = ";
    print(c1);
    cout << "c2 = ";
    print(c2);
    cout << "c3 = ";
    print(c3);
    cout << "c4 = ";
    print(c4);

    where_iter = c2.begin();
    ++where_iter; // start at second element
    c2.splice(where_iter, c1);
    cout << "After splicing c1 into c2:" << endl;
    cout << "c1 = ";
    print(c1);
    cout << "c2 = ";
    print(c2);

    first_iter = c3.begin();
    c2.splice(where_iter, c3, first_iter);
    cout << "After splicing the first element of c3 into c2:" << endl;
    cout << "c3 = ";
    print(c3);
    cout << "c2 = ";
    print(c2);

    first_iter = c4.begin();
    last_iter = c4.end();
    // set up to get the middle elements
    ++first_iter;
    --last_iter;
    c2.splice(where_iter, c4, first_iter, last_iter);
    cout << "After splicing a range of c4 into c2:" << endl;
    cout << "c4 = ";
    print(c4);
    cout << "c2 = ";
    print(c2);
}
```

Output

```
Beginning state of lists:c1 = 2 elements: (10) (11)c2 = 3 elements: (20) (21) (22)
c3 = 2 elements: (30) (31)c4 = 4 elements: (40) (41) (42) (43)After splicing c1
into c2:c1 = 0 elements:c2 = 5 elements: (20) (10) (11) (21) (22)After splicing the
first element of c3 into c2:c3 = 1 elements: (31)c2 = 6 elements: (20) (10) (11)
(30) (21) (22)After splicing a range of c4 into c2:c4 = 2 elements: (40) (43)c2 = 8
elements: (20) (10) (11) (30) (41) (42) (21) (22)
```

list::swap

Exchanges the elements of two lists.

unstlib

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

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

Parameters

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

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

Example

```
// list_swap.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

int main( )
{
   using namespace std;
   list <int> c1, c2, c3;
   list <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 list 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, list 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, list c1 is:";
   for ( c1_Iter = c1.begin( ); c1_Iter != c1.end( ); c1_Iter++ )
      cout << " " << *c1_Iter;
   cout << endl;
}
```

Output

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

list::unique

Removes adjacent duplicate elements or adjacent elements that satisfy some other binary predicate from a list.

```
void unique();

template <class BinaryPredicate>
void unique(BinaryPredicate pred);
```

Parameters

`pred`
The binary predicate used to compare successive elements.

Remarks

This function assumes that the list is sorted, so that all duplicate elements are adjacent. Duplicates that are not adjacent will not be deleted.

The first member function removes every element that compares equal to its preceding element.

The second member function removes every element that satisfies the predicate function `pred` when compared with its preceding element. You can use any of the binary function objects declared in the `<functional>`header for the argument pred or you can create your own.

Example

```
// list_unique.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

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

```
    list <int>::iterator c1_Iter, c2_Iter,c3_Iter;
    not_equal_to<int> mypred;

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

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

    list <int> c2 = c1;
    c2.unique( );
    cout << "After removing successive duplicate elements, c2 =";
    for ( c2_Iter = c2.begin( ); c2_Iter != c2.end( ); c2_Iter++ )
       cout << " " << *c2_Iter;
    cout << endl;

    list <int> c3 = c2;
    c3.unique( mypred );
    cout << "After removing successive unequal elements, c3 =";
    for ( c3_Iter = c3.begin( ); c3_Iter != c3.end( ); c3_Iter++ )
       cout << " " << *c3_Iter;
    cout << endl;
 }
```

Output

```
The initial list is c1 = -10 10 10 20 20 -10
After removing successive duplicate elements, c2 = -10 10 20 -10
After removing successive unequal elements, c3 = -10 -10
```

list::value_type

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

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

Remarks

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

Example

```
// list_value_type.cpp
// compile with: /EHsc
#include <list>
#include <iostream>

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

Output

```
44
```

See Also

<list>
Thread Safety in the C++ Standard Library
Standard Template Library