set 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 set Class.

The STL container class set is used for the storage and retrieval of data from a collection in which the values of the elements contained are unique and serve as the key values according to which the data is automatically ordered. The value of an element in a set may not be changed directly. Instead, you must delete old values and insert elements with new values.

Syntax

```
template <class Key,
    class Traits=less<Key>,
    class Allocator=allocator<Key>>
class set
```

Parameters

`Key`
The element data type to be stored in the set.

`Traits`
The type that provides a function object that can compare two element values as sort keys to determine their relative order in the set. This argument is optional, and the binary predicate **less** <*Key*> is the default value.

In C++14 you can enable heterogeneous lookup by specifying the `std::less<>` or `std::greater<>` predicate that has no type parameters. For more information, see Heterogeneous Lookup in Associative Containers

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

Remarks

An STL set is:

- An associative container, which a variable size container that supports the efficient retrieval of element values based on an associated key value. Further, it is a simple associative container because its element values are its key values.

- Reversible, because it provides a bidirectional iterator to access its elements.
- Sorted, because its elements are ordered by key values within the container in accordance with a specified comparison function.
- Unique in the sense that each of its elements must have a unique key. Since set is also a simple associative container, its elements are also unique.

A set is also described as a template class because the functionality it provides is generic and independent of the specific type of data contained as elements. The data type to be used is, instead, specified as a parameter in the class template along with the comparison function and allocator.

The choice of container type should be based in general on the type of searching and inserting required by the application. Associative containers are optimized for the operations of lookup, insertion and removal. The member functions that explicitly support these operations are efficient, performing them in a time that is on average proportional to the logarithm of the number of elements in the container. Inserting elements invalidates no iterators, and removing elements invalidates only those iterators that had specifically pointed at the removed elements.

The set should be the associative container of choice when the conditions associating the values with their keys are satisfied by the application. The elements of a set are unique and serve as their own sort keys. A model for this type of structure is an ordered list of, say, words in which the words may occur only once. If multiple occurrences of the words were allowed, then a multiset would be the appropriate container structure. If values need to be attached to a list of unique key words, then a map would be an appropriate structure to contain this data. If instead the keys are not unique, then a multimap would be the container of choice.

The set orders the sequence it controls by calling a stored function object of type key_compare. This stored object is a comparison function that may be accessed by calling the member function key_comp. In general, the elements need to be merely less than comparable to establish this order so that, given any two elements, it may be determined either that they are equivalent (in the sense that neither is less than the other) or that one is less than the other. This results in an ordering between the nonequivalent elements. On a more technical note, the comparison function is a binary predicate that induces a strict weak ordering in the standard mathematical sense. A binary predicate *f*(*x*,*y*) is a function object that has two argument objects *x* and *y* and a return value of **true** or **false**. An ordering imposed on a set is a strict weak ordering if the binary predicate is irreflexive, antisymmetric, and transitive and if equivalence is transitive, where two objects *x* and *y* are defined to be equivalent when both *f* (*x*,*y*) and *f*(*y*,*x*) are false. If the stronger condition of equality between keys replaces that of equivalence, then the ordering becomes total (in the sense that all the elements are ordered with respect to each other) and the keys matched will be indiscernible from each other.

In C++14 you can enable heterogeneous lookup by specifying the `std::less<>` or `std::greater<>` predicate that has no type parameters. For more information, see Heterogeneous Lookup in Associative Containers

The iterator provided by the set class is a bidirectional iterator, but the class member functions insert and set have versions that take as template parameters a weaker input iterator, whose functionality requirements are more minimal than those guaranteed by the class of bidirectional iterators. The different iterator concepts form a family related by refinements in their functionality. Each iterator concept has its own set of requirements, and the algorithms that work with them must limit their assumptions to the requirements provided by that type of iterator. It may be assumed that an input iterator may be dereferenced to refer to some object and that it may be incremented to the next iterator in the sequence. This is a minimal set of functionality, but it is enough to be able to talk meaningfully about a range of iterators [`First`, `Last`) in the context of the class's member functions.

Constructors

set	Constructs a set that is empty or that is a copy of all or part of some other set.

Typedefs

allocator_type	A type that represents the `allocator` class for the set object.
const_iterator	A type that provides a bidirectional iterator that can read a `const` element in the set.
const_pointer	A type that provides a pointer to a `const` element in a set.
const_reference	A type that provides a reference to a `const` element stored in a set for reading and performing `const` operations.
const_reverse_iterator	A type that provides a bidirectional iterator that can read any `const` element in the set.
difference_type	A signed integer type that can be used to represent the number of elements of a set in a range between elements pointed to by iterators.
iterator	A type that provides a bidirectional iterator that can read or modify any element in a set.
key_compare	A type that provides a function object that can compare two sort keys to determine the relative order of two elements in the set.
key_type	The type describes an object stored as an element of a set in its capacity as sort key.
pointer	A type that provides a pointer to an element in a set.
reference	A type that provides a reference to an element stored in a set.
reverse_iterator	A type that provides a bidirectional iterator that can read or modify an element in a reversed set.
size_type	An unsigned integer type that can represent the number of elements in a set.
value_compare	The type that provides a function object that can compare two elements to determine their relative order in the set.
value_type	The type describes an object stored as an element of a set in its capacity as a value.

Member Functions

begin	Returns an iterator that addresses the first element in the set.
cbegin	Returns a const iterator that addresses the first element in the set.
cend	Returns a const iterator that addresses the location succeeding the last element in a set.
clear	Erases all the elements of a set.
count	Returns the number of elements in a set whose key matches a parameter-specified key.
crbegin	Returns a const iterator addressing the first element in a reversed set.
crend	Returns a const iterator that addresses the location succeeding the last element in a reversed set.
emplace	Inserts an element constructed in place into a set.
emplace_hint	Inserts an element constructed in place into a set, with a placement hint.
empty	Tests if a set is empty.
end	Returns an iterator that addresses the location succeeding the last element in a set.
equal_range	Returns a pair of iterators respectively to the first element in a set with a key that is greater than a specified key and to the first element in the set with a key that is equal to or greater than the key.
erase	Removes an element or a range of elements in a set from specified positions or removes elements that match a specified key.
find	Returns an iterator addressing the location of an element in a set that has a key equivalent to a specified key.
get_allocator	Returns a copy of the `allocator` object used to construct the set.
insert	Inserts an element or a range of elements into a set.
key_comp	Retrieves a copy of the comparison object used to order keys in a set.
lower_bound	Returns an iterator to the first element in a set with a key that is equal to or greater than a specified key.
max_size	Returns the maximum length of the set.
rbegin	Returns an iterator addressing the first element in a reversed set.
rend	Returns an iterator that addresses the location succeeding the last element in a reversed set.
size	Returns the number of elements in the set.

swap	Exchanges the elements of two sets.
upper_bound	Returns an iterator to the first element in a set with a key that is greater than a specified key.
value_comp	Retrieves a copy of the comparison object used to order element values in a set.

Operators

operator=	Replaces the elements of a set with a copy of another set.

Requirements

Header: <set>

Namespace: std

set::allocator_type

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

```
typedef Allocator allocator_type;
```

Remarks

allocator_type is a synonym for the template parameter Allocator.

Returns the function object that a multiset uses to order its elements, which is the template parameter `Allocator`.

For more information on `Allocator`, see the Remarks section of the set Class topic.

Example

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

set::begin

Returns an iterator that addresses the first element in the set.

```
const_iterator begin() const;

iterator begin();
```

Return Value

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

Remarks

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

Example

```
// set_begin.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int> s1;
   set <int>::iterator s1_Iter;
   set <int>::const_iterator s1_cIter;

   s1.insert( 1 );
   s1.insert( 2 );
   s1.insert( 3 );

   s1_Iter = s1.begin( );
   cout << "The first element of s1 is " << *s1_Iter << endl;

   s1_Iter = s1.begin( );
   s1.erase( s1_Iter );

   // The following 2 lines would err because the iterator is const
   // s1_cIter = s1.begin( );
   // s1.erase( s1_cIter );

   s1_cIter = s1.begin( );
   cout << "The first element of s1 is now " << *s1_cIter << endl;
}
```

Output

```
The first element of s1 is 1
The first element of s1 is now 2
```

set::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++

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

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

set::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.

set::clear

Erases all the elements of a set.

```
void clear();
```

Example

```
// set_clear.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int> s1;

   s1.insert( 1 );
   s1.insert( 2 );

   cout << "The size of the set is initially " << s1.size( )
        << "." << endl;
```

```
   s1.clear( );
   cout << "The size of the set after clearing is "
        << s1.size( ) << "." << endl;
}
```

Output

```
The size of the set is initially 2.
The size of the set after clearing is 0.
```

set::const_iterator

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

```
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 begin for an example that uses `const_iterator`.

set::const_pointer

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

```
typedef typename allocator_type::const_pointer const_pointer;
```

Remarks

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

In most cases, a const_iterator should be used to access the elements in a const set object.

set::const_reference

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

```
typedef typename allocator_type::const_reference const_reference;
```

Example

```
// set_const_ref.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int> s1;

   s1.insert( 10 );
   s1.insert( 20 );

   // Declare and initialize a const_reference &Ref1
   // to the 1st element
   const int &Ref1 = *s1.begin( );

   cout << "The first element in the set is "
        << Ref1 << "." << endl;

   // The following line would cause an error because the
   // const_reference cannot be used to modify the set
   // Ref1 = Ref1 + 5;
}
```

Output

```
The first element in the set is 10.
```

set::const_reverse_iterator

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

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

Remarks

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

Example

See the example for rend for an example of how to declare and use the `const_reverse_iterator`.

set::count

Returns the number of elements in a set whose key matches a parameter-specified key.

```
size_type count(const Key& key) const;
```

Parameters

key
The key of the elements to be matched from the set.

Return Value

1 if the set contains an element whose sort key matches the parameter key. 0 if the set does not contain an element with a matching key.

Remarks

The member function returns the number of elements in the following range:

[`lower_bound` (_ *Key*), `upper_bound` (_ *Key*)).

Example

The following example demonstrates the use of the set::count member function.

```
// set_count.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main()
{
    using namespace std;
    set<int> s1;
    set<int>::size_type i;

    s1.insert(1);
    s1.insert(1);
```

```
    // Keys must be unique in set, so duplicates are ignored
    i = s1.count(1);
    cout << "The number of elements in s1 with a sort key of 1 is: "
         << i << "." << endl;

    i = s1.count(2);
    cout << "The number of elements in s1 with a sort key of 2 is: "
         << i << "." << endl;
}
```

Output

```
The number of elements in s1 with a sort key of 1 is: 1.
The number of elements in s1 with a sort key of 2 is: 0.
```

set::crbegin

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

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

Return Value

A const reverse bidirectional iterator addressing the first element in a reversed set or addressing what had been the last element in the unreversed set.

Remarks

`crbegin` is used with a reversed set just as begin is used with a set.

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

Example

```
// set_crbegin.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int> s1;
   set <int>::const_reverse_iterator s1_crIter;
```

```
   s1.insert( 10 );
   s1.insert( 20 );
   s1.insert( 30 );

   s1_crIter = s1.crbegin( );
   cout << "The first element in the reversed set is "
        << *s1_crIter << "." << endl;
}
```

Output

```
The first element in the reversed set is 30.
```

set::crend

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

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

Return Value

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

Remarks

`crend` is used with a reversed set just as end is used with a set.

With the return value of `crend`, the set object cannot be modified. The value returned by `crend` should not be dereferenced.

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

Example

```
// set_crend.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main() {
   using namespace std;
   set <int> s1;
```

```
   set <int>::const_reverse_iterator s1_crIter;

   s1.insert( 10 );
   s1.insert( 20 );
   s1.insert( 30 );

   s1_crIter = s1.crend( );
   s1_crIter--;
   cout << "The last element in the reversed set is "
        << *s1_crIter << "." << endl;
}
```

set::difference_type

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

```
typedef typename allocator_type::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 such as set, subtraction between iterators is only supported by random-access iterators provided by a random-access container such as vector.

Example

```
// set_diff_type.cpp
// compile with: /EHsc
#include <iostream>
#include <set>
#include <algorithm>

int main( )
{
   using namespace std;

   set <int> s1;
   set <int>::iterator s1_Iter, s1_bIter, s1_eIter;
```

```
    s1.insert( 20 );
    s1.insert( 10 );
    s1.insert( 20 );   // won't insert as set elements are unique

    s1_bIter = s1.begin( );
    s1_eIter = s1.end( );

    set <int>::difference_type   df_typ5, df_typ10, df_typ20;

    df_typ5 = count( s1_bIter, s1_eIter, 5 );
    df_typ10 = count( s1_bIter, s1_eIter, 10 );
    df_typ20 = count( s1_bIter, s1_eIter, 20 );

    // the keys, and hence the elements of a set are unique,
    // so there is at most one of a given value
    cout << "The number '5' occurs " << df_typ5
         << " times in set s1.\n";
    cout << "The number '10' occurs " << df_typ10
         << " times in set s1.\n";
    cout << "The number '20' occurs " << df_typ20
         << " times in set s1.\n";

    // count the number of elements in a set
    set <int>::difference_type  df_count = 0;
    s1_Iter = s1.begin( );
    while ( s1_Iter != s1_eIter)
    {
       df_count++;
       s1_Iter++;
    }

    cout << "The number of elements in the set s1 is: "
         << df_count << "." << endl;
}
```

Output

```
The number '5' occurs 0 times in set s1.
The number '10' occurs 1 times in set s1.
The number '20' occurs 1 times in set s1.
The number of elements in the set s1 is: 2.
```

set::emplace

Inserts an element constructed in place (no copy or move operations are performed).

```
template <class... Args>
pair<iterator, bool>
emplace(
    Args&&... args);
```

Parameters

Parameter	Description
`args`	The arguments forwarded to construct an element to be inserted into the set unless it already contains an element whose value is equivalently ordered.

Return Value

A pair whose bool component returns true if an insertion was made, and false if the map already contained an element whose value had an equivalent value in the ordering. The iterator component of the return value pair returns the address where a new element was inserted (if the bool component is true) or where the element was already located (if the bool component is false).

Remarks

No iterators or references are invalidated by this function.

During emplacement, if an exception is thrown, the container's state is not modified.

Example

C++

```
// set_emplace.cpp
// compile with: /EHsc
#include <set>
#include <string>
#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()
{
```

```
    set<string> s1;

    auto ret = s1.emplace("ten");

    if (!ret.second){
        cout << "Emplace failed, element with value \"ten\" already exists."
            << endl << "  The existing element is (" << *ret.first << ")"
            << endl;
        cout << "set not modified" << endl;
    }
    else{
        cout << "set modified, now contains ";
        print(s1);
    }
    cout << endl;

    ret = s1.emplace("ten");

    if (!ret.second){
        cout << "Emplace failed, element with value \"ten\" already exists."
            << endl << "  The existing element is (" << *ret.first << ")"
            << endl;
    }
    else{
        cout << "set modified, now contains ";
        print(s1);
    }
    cout << endl;
}
```

set::emplace_hint

Inserts an element constructed in place (no copy or move operations are performed), with a placement hint.

```
template <class... Args>
iterator emplace_hint(
    const_iterator where,
    Args&&... args);
```

Parameters

Parameter	Description
args	

	The arguments forwarded to construct an element to be inserted into the set unless the set already contains that element or, more generally, unless it already contains an element whose value is equivalently ordered.
`where`	The place to start searching for the correct point of insertion. (If that point immediately precedes `where`, insertion can occur in amortized constant time instead of logarithmic time.)

Return Value

An iterator to the newly inserted element.

If the insertion failed because the element already exists, returns an iterator to the existing element.

Remarks

No iterators or references are invalidated by this function.

During emplacement, if an exception is thrown, the container's state is not modified.

Example

C++

```
// set_emplace.cpp
// compile with: /EHsc
#include <set>
#include <string>
#include <iostream>

using namespace std;

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

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

    cout << endl;
}

int main()
{
    set<string> s1;

    // Emplace some test data
    s1.emplace("Anna");
    s1.emplace("Bob");
    s1.emplace("Carmine");

    cout << "set starting data: ";
    print(s1);
    cout << endl;
```

```
    // Emplace with hint
    // s1.end() should be the "next" element after this emplacement
    s1.emplace_hint(s1.end(), "Doug");

    cout << "set modified, now contains ";
    print(s1);
    cout << endl;
}
```

set::empty

Tests if a set is empty.

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

Return Value

true if the set is empty; **false** if the set is nonempty.

Example

```
// set_empty.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int> s1, s2;
   s1.insert ( 1 );

   if ( s1.empty( ) )
      cout << "The set s1 is empty." << endl;
   else
      cout << "The set s1 is not empty." << endl;

   if ( s2.empty( ) )
      cout << "The set s2 is empty." << endl;
   else
      cout << "The set s2 is not empty." << endl;
}
```

Output

```
The set s1 is not empty.
The set s2 is empty.
```

set::end

Returns the past-the-end iterator.

```
const_iterator end() const;


iterator end();
```

Return Value

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

Remarks

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

The value returned by **end** should not be dereferenced.

For a code example, see set::find.

set::equal_range

Returns a pair of iterators respectively to the first element in a set with a key that is greater than or equal to a specified key and to the first element in the set with a key that is greater than the key.

```
pair <const_iterator, const_iterator> equal_range (const Key& key) const;

pair <iterator, iterator> equal_range (const Key& key);
```

Parameters

key
The argument key to be compared with the sort key of an element from the set being searched.

Return Value

A pair of iterators where the first is the lower_bound of the key and the second is the upper_bound of the key.

To access the first iterator of a pair `pr` returned by the member function, use `pr`. **first**, and to dereference the lower bound iterator, use *(`pr`. **first**). To access the second iterator of a pair `pr` returned by the member function, use `pr`. **second**, and to dereference the upper bound iterator, use *(`pr`. **second**).

Example

```
// set_equal_range.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   typedef set<int, less< int > > IntSet;
   IntSet s1;
   set <int, less< int > > :: const_iterator s1_RcIter;

   s1.insert( 10 );
   s1.insert( 20 );
   s1.insert( 30 );

   pair <IntSet::const_iterator, IntSet::const_iterator> p1, p2;
   p1 = s1.equal_range( 20 );

   cout << "The upper bound of the element with "
        << "a key of 20 in the set s1 is: "
        << *(p1.second) << "." << endl;

   cout << "The lower bound of the element with "
        << "a key of 20 in the set s1 is: "
        << *(p1.first) << "." << endl;

   // Compare the upper_bound called directly
   s1_RcIter = s1.upper_bound( 20 );
   cout << "A direct call of upper_bound( 20 ) gives "
        << *s1_RcIter << "," << endl
        << "matching the 2nd element of the pair"
        << " returned by equal_range( 20 )." << endl;

   p2 = s1.equal_range( 40 );

   // If no match is found for the key,
   // both elements of the pair return end( )
   if ( ( p2.first == s1.end( ) ) && ( p2.second == s1.end( ) ) )
      cout << "The set s1 doesn't have an element "
           << "with a key less than 40." << endl;
   else
      cout << "The element of set s1 with a key >= 40 is: "
```

```
            << *(p1.first) << "." << endl;
}
```

Output

```
The upper bound of the element with a key of 20 in the set s1 is: 30.
The lower bound of the element with a key of 20 in the set s1 is: 20.
A direct call of upper_bound( 20 ) gives 30,
matching the 2nd element of the pair returned by equal_range( 20 ).
The set s1 doesn't have an element with a key less than 40.
```

set::erase

Removes an element or a range of elements in a set from specified positions or removes elements that match a specified key.

```
iterator erase(
    const_iterator Where);

iterator erase(
    const_iterator First,
    const_iterator Last);

size_type erase(
    const key_type& Key);
```

Parameters

`Where`
Position of the element to be removed.

`First`
Position of the first element to be removed.

`Last`
Position just beyond the last element to be removed.

`Key`
The key value of the elements to be removed.

Return Value

For the first two member functions, a bidirectional iterator that designates the first element remaining beyond any elements removed, or an element that is the end of the set if no such element exists.

For the third member function, returns the number of elements that have been removed from the set.

Remarks

Example

C++

```
// set_erase.cpp
// compile with: /EHsc
#include <set>
#include <string>
#include <iostream>
#include <iterator> // next() and prev() helper functions

using namespace std;

using myset = set<string>;

void printset(const myset& s) {
    for (const auto& iter : s) {
        cout << " [" << iter << "]";
    }
    cout << endl << "size() == " << s.size() << endl << endl;
}

int main()
{
    myset s1;

    // Fill in some data to test with, one at a time
    s1.insert("Bob");
    s1.insert("Robert");
    s1.insert("Bert");
    s1.insert("Rob");
    s1.insert("Bobby");

    cout << "Starting data of set s1 is:" << endl;
    printset(s1);
    // The 1st member function removes an element at a given position
    s1.erase(next(s1.begin()));
    cout << "After the 2nd element is deleted, the set s1 is:" << endl;
    printset(s1);

    // Fill in some data to test with, one at a time, using an intializer list
    myset s2{ "meow", "hiss", "purr", "growl", "yowl" };

    cout << "Starting data of set s2 is:" << endl;
    printset(s2);
    // The 2nd member function removes elements
    // in the range [First, Last)
    s2.erase(next(s2.begin()), prev(s2.end()));
    cout << "After the middle elements are deleted, the set s2 is:" << endl;
    printset(s2);

    myset s3;
```

```
    // Fill in some data to test with, one at a time, using emplace
    s3.emplace("C");
    s3.emplace("C#");
    s3.emplace("D");
    s3.emplace("D#");
    s3.emplace("E");
    s3.emplace("E#");
    s3.emplace("F");
    s3.emplace("F#");
    s3.emplace("G");
    s3.emplace("G#");
    s3.emplace("A");
    s3.emplace("A#");
    s3.emplace("B");

    cout << "Starting data of set s3 is:" << endl;
    printset(s3);
    // The 3rd member function removes elements with a given Key
    myset::size_type count = s3.erase("E#");
    // The 3rd member function also returns the number of elements removed
    cout << "The number of elements removed from s3 is: " << count << "." <<
endl;
    cout << "After the element with a key of \"E#\" is deleted, the set s3 is:" <<
endl;
    printset(s3);
}
```

set::find

Returns an iterator that refers to the location of an element in a set that has a key equivalent to a specified key.

```
iterator find(const Key& key);


const_iterator find(const Key& key) const;
```

Parameters

key
The key value to be matched by the sort key of an element from the set being searched.

Return Value

An iterator that refers to the location of an element with a specified key, or the location succeeding the last element in the set (`set::end()`) if no match is found for the key.

Remarks

The member function returns an iterator that refers to an element in the set whose key is equivalent to the argument key under a binary predicate that induces an ordering based on a less than comparability relation.

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

Example

C++

```
// compile with: /EHsc /W4 /MTd
#include <set>
#include <iostream>
#include <vector>
#include <string>

using namespace std;

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

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

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

template <typename C, class T> void findit(const C& c, T val) {
    cout << "Trying find() on value " << val << endl;
    auto result = c.find(val);
    if (result != c.end()) {
        cout << "Element found: "; print_elem(*result); cout << endl;
    } else {
        cout << "Element not found." << endl;
    }
}

int main()
{
    set<int> s1({ 40, 45 });
    cout << "The starting set s1 is: " << endl;
    print_collection(s1);

    vector<int> v;
    v.push_back(43);
    v.push_back(41);
    v.push_back(46);
    v.push_back(42);
    v.push_back(44);
    v.push_back(44); // attempt a duplicate
```

```
    cout << "Inserting the following vector data into s1: " << endl;
    print_collection(v);

    s1.insert(v.begin(), v.end());

    cout << "The modified set s1 is: " << endl;
    print_collection(s1);
    cout << endl;
    findit(s1, 45);
    findit(s1, 6);
}
```

set::get_allocator

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

```
allocator_type get_allocator() const;
```

Return Value

The allocator used by the set to manage memory, which is the template parameter `Allocator`.

For more information on `Allocator`, see the Remarks section of the set Class topic.

Remarks

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

Example

```
// set_get_allocator.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int>::allocator_type s1_Alloc;
   set <int>::allocator_type s2_Alloc;
   set <double>::allocator_type s3_Alloc;
   set <int>::allocator_type s4_Alloc;
```

```
   // The following lines declare objects
   // that use the default allocator.
   set <int> s1;
   set <int, allocator<int> > s2;
   set <double, allocator<double> > s3;

   s1_Alloc = s1.get_allocator( );
   s2_Alloc = s2.get_allocator( );
   s3_Alloc = s3.get_allocator( );

   cout << "The number of integers that can be allocated"
        << endl << "before free memory is exhausted: "
        << s2.max_size( ) << "." << endl;

   cout << "\nThe number of doubles that can be allocated"
        << endl << "before free memory is exhausted: "
        << s3.max_size( ) <<  "." << endl;

   // The following line creates a set s4
   // with the allocator of multiset s1.
   set <int> s4( less<int>( ), s1_Alloc );

   s4_Alloc = s4.get_allocator( );

   // Two allocators are interchangeable if
   // storage allocated from each can be
   // deallocated by the other
   if( s1_Alloc == s4_Alloc )
   {
      cout << "\nThe allocators are interchangeable."
           << endl;
   }
   else
   {
      cout << "\nThe allocators are not interchangeable."
           << endl;
   }
}
```

set::insert

Inserts an element or a range of elements into a set.

```
// (1) single element
pair<iterator, bool> insert(
    const value_type& Val);


// (2) single element, perfect forwarded
```

```
template <class ValTy>
pair<iterator, bool>
insert(
    ValTy&& Val);


// (3) single element with hint
iterator insert(
    const_iterator Where,
    const value_type& Val);


// (4) single element, perfect forwarded, with hint
template <class ValTy>
iterator insert(
    const_iterator Where,
    ValTy&& Val);


// (5) range
template <class InputIterator>
void insert(
    InputIterator First,
    InputIterator Last);


// (6) initializer list
void insert(
    initializer_list<value_type>
IList);
```

Parameters

Parameter	Description
`Val`	The value of an element to be inserted into the set unless it already contains an element whose value is equivalently ordered.
`Where`	The place to start searching for the correct point of insertion. (If that point immediately precedes `Where`, insertion can occur in amortized constant time instead of logarithmic time.)
`ValTy`	Template parameter that specifies the argument type that the set can use to construct an element of value_type, and perfect-forwards `Val` as an argument.
`First`	The position of the first element to be copied.
`Last`	The position just beyond the last element to be copied.

`InputIterator`	Template function argument that meets the requirements of an input iterator that points to elements of a type that can be used to construct value_type objects.
`IList`	The initializer_list from which to copy the elements.

Return Value

The single-element member functions, (1) and (2), return a pair whose `bool` component is true if an insertion was made, and false if the set already contained an element of equivalent value in the ordering. The iterator component of the return-value pair points to the newly inserted element if the `bool` component is true, or to the existing element if the `bool` component is false.

The single-element-with-hint member functions, (3) and (4), return an iterator that points to the position where the new element was inserted into the set or, if an element with an equivalent key already exists, to the existing element.

Remarks

No iterators, pointers, or references are invalidated by this function.

During the insertion of just one element, if an exception is thrown, the container's state is not modified. During the insertion of multiple elements, if an exception is thrown, the container is left in an unspecified but valid state.

To access the iterator component of a `pair``pr` that's returned by the single-element member functions, use `pr.first`; to dereference the iterator within the returned pair, use `*pr.first`, giving you an element. To access the `bool` component, use `pr.second`. For an example, see the sample code later in this article.

The value_type of a container is a typedef that belongs to the container, and, for set, `set<V>::value_type` is type `const V`.

The range member function (5) inserts the sequence of element values into a set that corresponds to each element addressed by an iterator in the range `[First, Last)`; therefore, `Last` does not get inserted. The container member function `end()` refers to the position just after the last element in the container—for example, the statement `s.insert(v.begin(), v.end());` attempts to insert all elements of `v` into `s`. Only elements that have unique values in the range are inserted; duplicates are ignored. To observe which elements are rejected, use the single-element versions of `insert`.

The initializer list member function (6) uses an initializer_list to copy elements into the set.

For insertion of an element constructed in place—that is, no copy or move operations are performed—see set::emplace and set::emplace_hint.

Example

C++

```
// set_insert.cpp
// compile with: /EHsc
#include <set>
#include <iostream>
#include <string>
#include <vector>
```

```
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()
{

    // insert single values
    set<int> s1;
    // call insert(const value_type&) version
    s1.insert({ 1, 10 });
    // call insert(ValTy&&) version
    s1.insert(20);

    cout << "The original set values of s1 are:" << endl;
    print(s1);

    // intentionally attempt a duplicate, single element
    auto ret = s1.insert(1);
    if (!ret.second){
        auto elem = *ret.first;
        cout << "Insert failed, element with value 1 already exists."
            << endl << "  The existing element is (" << elem << ")"
            << endl;
    }
    else{
        cout << "The modified set values of s1 are:" << endl;
        print(s1);
    }
    cout << endl;

    // single element, with hint
    s1.insert(s1.end(), 30);
    cout << "The modified set values of s1 are:" << endl;
    print(s1);
    cout << endl;

    // The templatized version inserting a jumbled range
    set<int> s2;
    vector<int> v;
    v.push_back(43);
    v.push_back(294);
    v.push_back(41);
    v.push_back(330);
    v.push_back(42);
    v.push_back(45);
```

```
    cout << "Inserting the following vector data into s2:" << endl;
    print(v);

    s2.insert(v.begin(), v.end());

    cout << "The modified set values of s2 are:" << endl;
    print(s2);
    cout << endl;

    // The templatized versions move-constructing elements
    set<string>  s3;
    string str1("blue"), str2("green");

    // single element
    s3.insert(move(str1));
    cout << "After the first move insertion, s3 contains:" << endl;
    print(s3);

    // single element with hint
    s3.insert(s3.end(), move(str2));
    cout << "After the second move insertion, s3 contains:" << endl;
    print(s3);
    cout << endl;

    set<int> s4;
    // Insert the elements from an initializer_list
    s4.insert({ 4, 44, 2, 22, 3, 33, 1, 11, 5, 55 });
    cout << "After initializer_list insertion, s4 contains:" << endl;
    print(s4);
    cout << endl;
}
```

set::iterator

A type that provides a constant bidirectional iterator that can read any element in a set.

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

Example

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

set::key_comp

Retrieves a copy of the comparison object used to order keys in a set.

```
key_compare key_comp() const;
```

Return Value

Returns the function object that a set uses to order its elements, which is the template parameter `Traits`.

For more information on `Traits` see the set Class topic.

Remarks

The stored object defines the member function:

bool operator()(**const Key&** _xVal, **const Key&** _yVal);

which returns **true** if _xVal precedes and is not equal to _yVal in the sort order.

Note that both key_compare and value_compare are synonyms for the template parameter **Traits**. Both types are provided for the set and multiset classes, where they are identical, for compatibility with the map and multimap classes, where they are distinct.

Example

```
// set_key_comp.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;

   set <int, less<int> > s1;
   set<int, less<int> >::key_compare kc1 = s1.key_comp( ) ;
   bool result1 = kc1( 2, 3 ) ;
   if( result1 == true )
   {
      cout << "kc1( 2,3 ) returns value of true, "
           << "where kc1 is the function object of s1."
           << endl;
   }
   else
   {
      cout << "kc1( 2,3 ) returns value of false "
           << "where kc1 is the function object of s1."
           << endl;
   }

   set <int, greater<int> > s2;
```

```
    set<int, greater<int> >::key_compare kc2 = s2.key_comp( ) ;
    bool result2 = kc2( 2, 3 ) ;
    if(result2 == true)
    {
       cout << "kc2( 2,3 ) returns value of true, "
            << "where kc2 is the function object of s2."
            << endl;
    }
    else
    {
       cout << "kc2( 2,3 ) returns value of false, "
            << "where kc2 is the function object of s2."
            << endl;
    }
}
```

Output

```
kc1( 2,3 ) returns value of true, where kc1 is the function object of s1.
kc2( 2,3 ) returns value of false, where kc2 is the function object of s2.
```

set::key_compare

A type that provides a function object that can compare two sort keys to determine the relative order of two elements in the set.

```
typedef Traits key_compare;
```

Remarks

`key_compare` is a synonym for the template parameter `Traits`.

For more information on `Traits` see the set Class topic.

Note that both `key_compare` and value_compare are synonyms for the template parameter **Traits**. Both types are provided for the set and multiset classes, where they are identical, for compatibility with the map and multimap classes, where they are distinct.

Example

See the example for key_comp for an example of how to declare and use `key_compare`.

set::key_type

A type that describes an object stored as an element of a set in its capacity as sort key.

```
typedef Key key_type;
```

Remarks

`key_type` is a synonym for the template parameter `Key`.

For more information on `Key`, see the Remarks section of the set Class topic.

Note that both `key_type` and value_type are synonyms for the template parameter **Key**. Both types are provided for the set and multiset classes, where they are identical, for compatibility with the map and multimap classes, where they are distinct.

Example

See the example for value_type for an example of how to declare and use `key_type`.

set::lower_bound

Returns an iterator to the first element in a set with a key that is equal to or greater than a specified key.

```
const_iterator lower_bound(const Key& key) const;

iterator lower_bound(const Key& key);
```

Parameters

key
The argument key to be compared with the sort key of an element from the set being searched.

Return Value

An iterator or `const_iterator` that addresses the location of an element in a set that with a key that is equal to or greater than the argument key or that addresses the location succeeding the last element in the set if no match is found for the key.

Example

```
// set_lower_bound.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
```

```
{
   using namespace std;
   set <int> s1;
   set <int> :: const_iterator s1_AcIter, s1_RcIter;

   s1.insert( 10 );
   s1.insert( 20 );
   s1.insert( 30 );

   s1_RcIter = s1.lower_bound( 20 );
   cout << "The element of set s1 with a key of 20 is: "
        << *s1_RcIter << "." << endl;

   s1_RcIter = s1.lower_bound( 40 );

   // If no match is found for the key, end( ) is returned
   if ( s1_RcIter == s1.end( ) )
      cout << "The set s1 doesn't have an element "
           << "with a key of 40." << endl;
   else
      cout << "The element of set s1 with a key of 40 is: "
           << *s1_RcIter << "." << endl;

   // The element at a specific location in the set can be found
   // by using a dereferenced iterator that addresses the location
   s1_AcIter = s1.end( );
   s1_AcIter--;
   s1_RcIter = s1.lower_bound( *s1_AcIter );
   cout << "The element of s1 with a key matching "
        << "that of the last element is: "
        << *s1_RcIter << "." << endl;
}
```

Output

```
The element of set s1 with a key of 20 is: 20.
The set s1 doesn't have an element with a key of 40.
The element of s1 with a key matching that of the last element is: 30.
```

set::max_size

Returns the maximum length of the set.

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

Return Value

The maximum possible length of the set.

Example

```
// set_max_size.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int> s1;
   set <int>::size_type i;

   i = s1.max_size( );
   cout << "The maximum possible length "
        << "of the set is " << i << "." << endl;
}
```

set::operator=

Replaces the elements of this `set` using elements from another `set`.

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

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

Parameters

Parameter	Description
`right`	The `set` providing new elements to be assigned to this `set`.

Remarks

The first version of `operator=` uses an lvalue reference for `right`, to copy elements from `right` to this `set`.

The second version uses an rvalue reference for right. It moves elements from `right` to this `set`.

Any elements in this set before the operator function executes are discarded.

Example

```
// set_operator_as.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

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

   v1.insert(10);

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

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

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

set::pointer

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

```
typedef typename allocator_type::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 set object.

set::rbegin

Returns an iterator addressing the first element in a reversed set.

```
const_reverse_iterator rbegin() const;

reverse_iterator rbegin();
```

Return Value

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

Remarks

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

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

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

Example

```
// set_rbegin.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int> s1;
   set <int>::iterator s1_Iter;
   set <int>::reverse_iterator s1_rIter;

   s1.insert( 10 );
   s1.insert( 20 );
   s1.insert( 30 );

   s1_rIter = s1.rbegin( );
   cout << "The first element in the reversed set is "
        << *s1_rIter << "." << endl;

   // begin can be used to start an iteration
   // throught a set in a forward order
```

```
   cout << "The set is:";
   for ( s1_Iter = s1.begin( ) ; s1_Iter != s1.end( ); s1_Iter++ )
      cout << " " << *s1_Iter;
   cout << endl;

   // rbegin can be used to start an iteration
   // throught a set in a reverse order
   cout << "The reversed set is:";
   for ( s1_rIter = s1.rbegin( ) ; s1_rIter != s1.rend( ); s1_rIter++ )
      cout << " " << *s1_rIter;
   cout << endl;

   // A set element can be erased by dereferencing to its key
   s1_rIter = s1.rbegin( );
   s1.erase ( *s1_rIter );

   s1_rIter = s1.rbegin( );
   cout << "After the erasure, the first element "
        << "in the reversed set is "<< *s1_rIter << "." << endl;
}
```

Output

```
The first element in the reversed set is 30.
The set is: 10 20 30
The reversed set is: 30 20 10
After the erasure, the first element in the reversed set is 20.
```

set::reference

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

```
typedef typename allocator_type::reference reference;
```

Example

```
// set_reference.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

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

```
    set <int> s1;

    s1.insert( 10 );
    s1.insert( 20 );

    // Declare and initialize a reference &Ref1 to the 1st element
    const int &Ref1 = *s1.begin( );

    cout << "The first element in the set is "
         << Ref1 << "." << endl;
}
```

Output

```
The first element in the set is 10.
```

set::rend

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

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

Remarks

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

If the return value of `rend` is assigned to a `const_reverse_iterator`, then the set object cannot be modified. If the return value of `rend` is assigned to a `reverse_iterator`, then the set object can be modified. The value returned by `rend` should not be dereferenced.

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

Example

```
// set_rend.cpp
// compile with: /EHsc
#include <set>
```

```
#include <iostream>

int main() {
   using namespace std;
   set <int> s1;
   set <int>::iterator s1_Iter;
   set <int>::reverse_iterator s1_rIter;
   set <int>::const_reverse_iterator s1_crIter;

   s1.insert( 10 );
   s1.insert( 20 );
   s1.insert( 30 );

   s1_rIter = s1.rend( );
   s1_rIter--;
   cout << "The last element in the reversed set is "
        << *s1_rIter << "." << endl;

   // end can be used to terminate an iteration
   // throught a set in a forward order
   cout << "The set is: ";
   for ( s1_Iter = s1.begin( ) ; s1_Iter != s1.end( ); s1_Iter++ )
      cout << *s1_Iter << " ";
   cout << "." << endl;

   // rend can be used to terminate an iteration
   // throught a set in a reverse order
   cout << "The reversed set is: ";
   for ( s1_rIter = s1.rbegin( ) ; s1_rIter != s1.rend( ); s1_rIter++ )
      cout << *s1_rIter << " ";
   cout << "." << endl;

   s1_rIter = s1.rend( );
   s1_rIter--;
   s1.erase ( *s1_rIter );

   s1_rIter = s1.rend( );
   --s1_rIter;
   cout << "After the erasure, the last element in the "
        << "reversed set is " << *s1_rIter << "." << endl;
}
```

set::reverse_iterator

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

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

Remarks

A type `reverse_iterator` is use to iterate through the set in reverse.

Example

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

set::set

Constructs a set that is empty or that is a copy of all or part of some other set.

```
set();

explicit set(
    const Traits& Comp);

set(
    const Traits& Comp,
    const Allocator& Al);

set(
    const set& Right);

set(
    set&& Right);

set(
    initializer_list<Type> IList);

set(
    initializer_list<Type> IList,
    const Compare& Comp);

set(
    initializer_list<Type> IList,
    const Compare& Comp,
    const Allocator& Al);


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

template <class InputIterator>
set(
 InputIterator First,
    InputIterator Last,
    const Traits& Comp);
```

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

Parameters

Parameter	Description
`Al`	The storage allocator class to be used for this set object, which defaults to **Allocator**.
`Comp`	The comparison function of type `const Traits` used to order the elements in the set, which defaults to `Compare`.
`Rght`	The set of which the constructed set 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 from which to copy the elements.

Remarks

All constructors store a type of allocator object that manages memory storage for the set and that can later be returned by calling get_allocator. The allocator parameter is often omitted in the class declarations and preprocessing macros used to substitute alternative allocators.

All constructors initialize their sets.

All constructors store a function object of type **Traits** that is used to establish an order among the keys of the set and that can later be returned by calling key_comp.

The first three constructors specify an empty initial set, the second specifying the type of comparison function (`comp`) to be used in establishing the order of the elements and the third explicitly specifying the allocator type (`al`) to be used. The keyword **explicit** suppresses certain kinds of automatic type conversion.

The fourth constructor specifies a copy of the set `right`.

The next three constructors use an initializer_list to specify the elements.

The next three constructors copy the range [`first`, `last`) of a set with increasing explicitness in specifying the type of comparison function of class **Traits** and **Allocator**.

The eighth constructor specifies a copy of the set by moving `right`.

Example

```
// set_set.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main()
{
    using namespace std;

    // Create an empty set s0 of key type integer
    set <int> s0;

    // Create an empty set s1 with the key comparison
    // function of less than, then insert 4 elements
    set <int, less<int> > s1;
    s1.insert(10);
    s1.insert(20);
    s1.insert(30);
    s1.insert(40);

    // Create an empty set s2 with the key comparison
    // function of less than, then insert 2 elements
    set <int, less<int> > s2;
    s2.insert(10);
    s2.insert(20);

    // Create a set s3 with the
    // allocator of set s1
    set <int>::allocator_type s1_Alloc;
    s1_Alloc = s1.get_allocator();
    set <int> s3(less<int>(), s1_Alloc);
    s3.insert(30);

    // Create a copy, set s4, of set s1
    set <int> s4(s1);

    // Create a set s5 by copying the range s1[ first,  last)
    set <int>::const_iterator s1_bcIter, s1_ecIter;
    s1_bcIter = s1.begin();
    s1_ecIter = s1.begin();
    s1_ecIter++;
    s1_ecIter++;
    set <int> s5(s1_bcIter, s1_ecIter);

    // Create a set s6 by copying the range s4[ first,  last)
    // and with the allocator of set s2
    set <int>::allocator_type s2_Alloc;
    s2_Alloc = s2.get_allocator();
    set <int> s6(s4.begin(), ++s4.begin(), less<int>(), s2_Alloc);

    cout << "s1 =";
    for (auto i : s1)
        cout << " " << i;
```

```
    cout << endl;

    cout << "s2 = " << *s2.begin() << " " << *++s2.begin() << endl;

    cout << "s3 =";
    for (auto i : s3)
        cout << " " << i;
    cout << endl;

    cout << "s4 =";
    for (auto i : s4)
        cout << " " << i;
    cout << endl;

    cout << "s5 =";
    for (auto i : s5)
        cout << " " << i;
    cout << endl;

    cout << "s6 =";
    for (auto i : s6)
        cout << " " << i;
    cout << endl;

    // Create a set by moving s5
    set<int> s7(move(s5));
    cout << "s7 =";
    for (auto i : s7)
        cout << " " << i;
    cout << endl;

    // Create a set with an initializer_list
    cout << "s8 =";
    set<int> s8{ { 1, 2, 3, 4 } };
    for (auto i : s8)
        cout << " " << i;
    cout << endl;

    cout << "s9 =";
    set<int> s9{ { 5, 6, 7, 8 }, less<int>() };
    for (auto i : s9)
        cout << " " << i;
    cout << endl;

    cout << "s10 =";
    set<int> s10{ { 10, 20, 30, 40 }, less<int>(), s9.get_allocator() };
    for (auto i : s10)
        cout << " " << i;
    cout << endl;
}
```

Output

```
s1 = 10 20 30 40s2 = 10 20s3 = 30s4 = 10 20 30 40s5 = 10 20s6 = 10s7 = 10 20s8 = 1
2 3 4s9 = 5 6 7 8s10 = 10 20 30 40
```

set::size

Returns the number of elements in the set.

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

Return Value

The current length of the set.

Example

```
// set_size.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int> s1;
   set <int> :: size_type i;

   s1.insert( 1 );
   i = s1.size( );
   cout << "The set length is " << i << "." << endl;

   s1.insert( 2 );
   i = s1.size( );
   cout << "The set length is now " << i << "." << endl;
}
```

Output

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

set::size_type

An unsigned integer type that can represent the number of elements in a set.

```
typedef typename allocator_type::size_type size_type;
```

Example

See the example for size for an example of how to declare and use `size_type`

set::swap

Exchanges the elements of two sets.

```
void swap(
    set<Key, Traits, Allocator>& right);
```

Parameters

`right`
The argument set providing the elements to be swapped with the target set.

Remarks

The member function invalidates no references, pointers, or iterators that designate elements in the two sets whose elements are being exchanged.

Example

```
// set_swap.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int> s1, s2, s3;
   set <int>::iterator s1_Iter;

   s1.insert( 10 );
   s1.insert( 20 );
```

```
   s1.insert( 30 );
   s2.insert( 100 );
   s2.insert( 200 );
   s3.insert( 300 );

   cout << "The original set s1 is:";
   for ( s1_Iter = s1.begin( ); s1_Iter != s1.end( ); s1_Iter++ )
      cout << " " << *s1_Iter;
   cout   << "." << endl;

   // This is the member function version of swap
   s1.swap( s2 );

   cout << "After swapping with s2, list s1 is:";
   for ( s1_Iter = s1.begin( ); s1_Iter != s1.end( ); s1_Iter++ )
      cout << " " << *s1_Iter;
   cout  << "." << endl;

   // This is the specialized template version of swap
   swap( s1, s3 );

   cout << "After swapping with s3, list s1 is:";
   for ( s1_Iter = s1.begin( ); s1_Iter != s1.end( ); s1_Iter++ )
      cout << " " << *s1_Iter;
   cout   << "." << endl;
}
```

Output

```
The original set s1 is: 10 20 30.
After swapping with s2, list s1 is: 100 200.
After swapping with s3, list s1 is: 300.
```

set::upper_bound

Returns an iterator to the first element in a set that with a key that is greater than a specified key.

```
const_iterator upper_bound(const Key& key) const;

iterator upper_bound(const Key& key);
```

Parameters

key
The argument key to be compared with the sort key of an element from the set being searched.

Return Value

An **iterator** or `const_iterator` that addresses the location of an element in a set that with a key that is greater than the argument key, or that addresses the location succeeding the last element in the set if no match is found for the key.

Example

```
// set_upper_bound.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int> s1;
   set <int> :: const_iterator s1_AcIter, s1_RcIter;

   s1.insert( 10 );
   s1.insert( 20 );
   s1.insert( 30 );

   s1_RcIter = s1.upper_bound( 20 );
   cout << "The first element of set s1 with a key greater "
        << "than 20 is: " << *s1_RcIter << "." << endl;

   s1_RcIter = s1.upper_bound( 30 );

   // If no match is found for the key, end( ) is returned
   if ( s1_RcIter == s1.end( ) )
      cout << "The set s1 doesn't have an element "
           << "with a key greater than 30." << endl;
   else
      cout << "The element of set s1 with a key > 40 is: "
           << *s1_RcIter << "." << endl;

   // The element at a specific location in the set can be found
   // by using a dereferenced iterator addressing the location
   s1_AcIter = s1.begin( );
   s1_RcIter = s1.upper_bound( *s1_AcIter );
   cout << "The first element of s1 with a key greater than"
        << endl << "that of the initial element of s1 is: "
        << *s1_RcIter << "." << endl;
}
```

Output

```
The first element of set s1 with a key greater than 20 is: 30.
The set s1 doesn't have an element with a key greater than 30.
The first element of s1 with a key greater than
```

```
that of the initial element of s1 is: 20.
```

set::value_comp

Retrieves a copy of the comparison object used to order element values in a set.

```
value_compare value_comp() const;
```

Return Value

Returns the function object that a set uses to order its elements, which is the template parameter `Traits`.

For more information on `Traits` see the set Class topic.

Remarks

The stored object defines the member function:

bool operator(**const Key&**_`xVal`, **const Key&**_`yVal`);

which returns **true** if _`xVal` precedes and is not equal to _`yVal` in the sort order.

Note that both value_compare and key_compare are synonyms for the template parameter **Traits**. Both types are provided for the set and multiset classes, where they are identical, for compatibility with the map and multimap classes, where they are distinct.

Example

```
// set_value_comp.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;

   set <int, less<int> > s1;
   set <int, less<int> >::value_compare vc1 = s1.value_comp( );
   bool result1 = vc1( 2, 3 );
   if( result1 == true )
   {
      cout << "vc1( 2,3 ) returns value of true, "
           << "where vc1 is the function object of s1."
           << endl;
   }
```

```
    else
    {
       cout << "vc1( 2,3 ) returns value of false, "
            << "where vc1 is the function object of s1."
            << endl;
    }

    set <int, greater<int> > s2;
    set<int, greater<int> >::value_compare vc2 = s2.value_comp( );
    bool result2 = vc2( 2, 3 );
    if( result2 == true )
    {
       cout << "vc2( 2,3 ) returns value of true, "
            << "where vc2 is the function object of s2."
            << endl;
    }
    else
    {
       cout << "vc2( 2,3 ) returns value of false, "
            << "where vc2 is the function object of s2."
            << endl;
    }
}
```

Output

```
vc1( 2,3 ) returns value of true, where vc1 is the function object of s1.
vc2( 2,3 ) returns value of false, where vc2 is the function object of s2.
```

set::value_compare

A type that provides a function object that can compare two element values to determine their relative order in the set.

```
typedef key_compare value_compare;
```

Remarks

`value_compare` is a synonym for the template parameter `Traits`.

For more information on `Traits` see the set Class topic.

Note that both key_compare and **value_compare** are synonyms for the template parameter **Traits**. Both types are provided for the set and multiset classes, where they are identical, for compatibility with the map and multimap classes, where they are distinct.

Example

See the example for value_comp for an example of how to declare and use `value_compare`.

set::value_type

A type that describes an object stored as an element of a set in its capacity as a value.

```
typedef Key value_type;
```

Remarks

`value_type` is a synonym for the template parameter `Key`.

For more information on `Key`, see the Remarks section of the set Class topic.

Note that both key_type and `value_type` are synonyms for the template parameter **Key**. Both types are provided for the set and multiset classes, where they are identical, for compatibility with the map and multimap classes, where they are distinct.

Example

```
// set_value_type.cpp
// compile with: /EHsc
#include <set>
#include <iostream>

int main( )
{
   using namespace std;
   set <int> s1;
   set <int>::iterator s1_Iter;

   set <int>::value_type svt_Int;   // Declare value_type
   svt_Int = 10;            // Initialize value_type

   set <int> :: key_type skt_Int;   // Declare key_type
   skt_Int = 20;             // Initialize key_type

   s1.insert( svt_Int );         // Insert value into s1
   s1.insert( skt_Int );         // Insert key into s1

   // A set accepts key_types or value_types as elements
   cout << "The set has elements:";
   for ( s1_Iter = s1.begin( ) ; s1_Iter != s1.end( ); s1_Iter++)
      cout << " " << *s1_Iter;
   cout << "." << endl;
```

```
}
```

Output

```
The set has elements: 10 20.
```

See Also

<set>
Containers
Thread Safety in the C++ Standard Library
Standard Template Library