**Sets and Multisets**

Set and multiset containers sort their elements automatically according to a certain sorting criterion. Multisets allow duplicates, whereas sets do not.

header file:

#include <set>

The types are defined as class templates inside namespace std:

namespace std {

template < typename T,

typename Compare = less<T>,

typename Allocator = allocator<T> >

class set;

template < typename T,

typename Compare = less<T>,

typename Allocator = allocator<T> >

class multiset;

}

The elements of a set or a multiset may have any type T that is comparable according to the sorting criterion. The default criterion less is used.

Default memory model is the model allocator, which is provided by the C++ STL.

The sorting criterion must define strict weak ordering, which is defined by the following four properties:

Sorting criterion has to be:

1. **antisymmetric:**

for operator <: If x < y is true, then y < x is false

for a predicate op(): If op(x,y) is true, then op(y,x) is false

1. **transitive:**

for operator <: If x < y is true and y < z is true, then x < z is true

for a predicate op(): If op(x,y) is true and op(y,z) is true, then op(x,z) is true

1. **irreflexive:**

for operator <: x < x is always false

for a predicate op(): op(x,x) is always false

1. **transitivity of equivalence:**

which means roughly: If a is equivalent to b and b is equivalent to c, then a is equivalent to c.

for operator <: If !(a<b) && !(b<a) is true and !(b<c) && !(c<b) is true then !(a<c) && !(c<a) is true.

for a predicate op(): If op(a,b), op(b,a), op(b,c), and op(c,b) all yield false, then op(a,c) and op(c,a) yield false.

The sorting criterion is also used to check equivalence.

That is, two elements are considered to be duplicates if neither is less than the other (or if both op(x,y) and op(y,x) are false).

For multisets, the order of equivalent elements is random but stable. Thus, insertions and erasures preserve the relative ordering of equivalent elements (guaranteed since C++11).

# Abilities of Sets and Multisets

Like all standardized associative container classes, sets and multisets are usually implemented as balanced binary trees. It follows from the complexity of set and multiset operations.

Adventage: searching. Search functions have logarithmic complexity.

Constraint on sets and multisets: You may not change the value of an element directly, because doing so might compromise the correct order.

Therefore, to modify the value of an element, you must remove the element having the old value and insert a new element that has the new value. The interface reflects this behavior:

* Sets and multisets don’t provide operations for direct element access.
* Indirect access via iterators has the constraint that, from the iterator’s point of view, the element value is constant.

# Set and Multiset Operations

## Create, Copy, and Destroy

|  |  |
| --- | --- |
| Operation | Effect |
| set c | Default constructor; creates an empty set/multiset without any elements |
| set c(op) | Creates an empty set/multiset that uses op as the sorting criterion |
| set c(c2) | Copy constructor; creates a copy of another set/multiset of the same type (all elements are copied) |
| set c = c2 | Copy constructor; creates a copy of another set/multiset of the same type (all elements are copied) |
| set c(rv) | Move constructor; creates a new set/multiset of the same type, taking the contents of the rvalue rv (since C++11) |
| set c = rv | Move constructor; creates a new set/multiset of the same type, taking the contents of the rvalue rv (since C++11) |
| set c(beg, end) | Creates a set/multiset initialized by the elements of the range [beg,end) |
| set c(beg, end, op) | Creates a set/multiset with the sorting criterion op initialized by the elements of the range [beg,end) |
| set c(initlist) | Creates a set/multiset initialized with the elements of initializer list initlist (since C++11) |
| set c = initlist | Creates a set/multiset initialized with the elements of initializer list initlist (since C++11) |
| c.~set() | Destroys all elements and frees the memory |

set may be one of the following types:

|  |  |
| --- | --- |
| Operation | Effect |
| set<Elem> | A set that by default sorts with less<> (operator <) |
| set<Elem, Op> | A set that by default sorts with Op |
| multiset<Elem> | A multiset that by default sorts with less<> (operator <) |
| multiset<Elem, Op> | A multiset that by default sorts with Op |

You can define the sorting criterion in two ways:

1. **As a template parameter**

std::set<int,std::greater<int>> coll;

Thus, the type system ensures that only containers with the same sorting criterion can be combined.

1. **As a constructor parameter**

In this case, you might have a type for several sorting criteria that allows having different initial values or states.

This is useful when processing the sorting criterion at runtime and when sorting criteria are needed that are different but of the same data type.

Default sorting criterion, function object less<>, is used

Sorting criterion is also used to check for equivalence of two elements in the same container (i.e., to find duplicates).

Equivalence of two elements looks like this:

if (! (elem1<elem2 || elem2<elem1))

This has three advantages:

1. You need to pass only one argument as the sorting criterion.
2. You don’t have to provide operator == for the element type.
3. You can have contrary definitions between equivalence and equality (however, this might be a source of confusion).

If two containers are compared by operator ==, the elements in both containers are compared using their operator ==, which means that operator == has to be provided for the element type.

## Nonmodifying Operations

nonmodifying operations to query the size and to make comparisons

|  |  |
| --- | --- |
| Operation | Effect |
| c.key\_comp() | Returns the comparison criterion |
| c.value\_comp() | Returns the comparison criterion for values as a whole (same as key\_comp()) |
| c.empty() | Returns whether the container is empty (equivalent to size()==0 but might be faster) |
| c.size() | Returns the current number of elements c.max\_size() Returns the maximum number of elements possible |
| c1 == c2 | Returns whether c1 is equal to c2 (calls == for the elements) |
| c1 != c2 | Returns whether c1 is not equal to c2 (equivalent to !(c1==c2)) |
| c1 < c2 | Returns whether c1 is less than c2 |
| c1 > c2 | Returns whether c1 is greater than c2 (equivalent to c2<c1) |
| c1 <= c2 | Returns whether c1 is less than or equal to c2 (equivalent to !(c2<c1)) |
| c1 >= c2 | Returns whether c1 is greater than or equal to c2 (equivalent to !(c1<c2)) |

Comparisons are provided only for containers of the same type.

Thus, the elements and the sorting criterion must have the same types; otherwise, a type error occurs at compile time.

std::set<float> c1; // sorting criterion: std::less<>

std::set<float,std::greater<float> > c2;

if (c1 == c2) { } // CE error: no match for 'operator==' (operand types are 'std::set<float>' and 'std::set<float, std::greater<float> >')

Comparison is is done by a lexicographical comparison. To compare containers of different types you must use the comparing algorithms.

## Special Search Operations

|  |  |
| --- | --- |
| Operation | Effect |
| c.count(val) | Returns the number of elements with value val |
| c.find(val) | Returns the position of the first element with value val (or end() if none found) |
| c.lower\_bound(val) | Returns the first position, where val would get inserted (the first element >= val) |
| c.upper\_bound(val) | Returns the last position, where val would get inserted (the first element > val) |
| c.equal\_range(val) | Returns a range with all elements with a value equal to |

Sets and multisets provide special optimized search functions. Always prefer the optimized versions for sets and multisets to achieve logarithmic complexity instead of the linear complexity of the general algorithms.

**find():** returns iterator position of the first element that has the value, or returns end() of the container if no such element is found

**lower\_bound() and upper\_bound():** return the first and last position, respectively, at which an element with the passed value would be inserted.

In other words,

lower\_bound() returns the position of the first element that has the same or a greater value than the argument

upper\_bound() returns the position of the first element with a greater value

**equal\_range():** returns both return values of lower\_bound() and upper\_bound() as a pair

if lower\_bound() == upper\_bound()

or

first value of equal\_range() == second value of equal\_range()

no elements with the same value exist in the set or multiset.

Naturally, the range of elements having the same values could contain at most one element in a set.

## Assignments

|  |  |
| --- | --- |
| Operation | Effect |
| c = c2 | Assigns all elements of c2 to c |
| c = rv | Move assigns all elements of the rvalue rv to c (since C++11) |
| c = initlist | Assigns all elements of the initializer list initlist to c (since C++11) |
| c1.swap(c2) | Swaps the data of c1 and c2 |
| swap(c1,c2) | Swaps the data of c1 and c2 |

For these operations, both containers must have the same type.

In particular, the type of the comparison criteria must be the same, although the comparison criteria themselves may be different.

If the criteria are different, they will also get assigned or swapped.

## Iterator Functions

|  |  |
| --- | --- |
| Operation | Effect |
| c.begin() | Returns a bidirectional iterator for the first element |
| c.end() | Returns a bidirectional iterator for the position after the last element |
| c.cbegin() | Returns a constant bidirectional iterator for the first element (since C++11) |
| c.cend() | Returns a constant bidirectional iterator for the position after the last element (since C++11) |
| c.rbegin() | Returns a reverse iterator for the first element of a reverse iteration |
| c.rend() | Returns a reverse iterator for the position after the last element of a reverse iteration |
| c.crbegin() | Returns a constant reverse iterator for the first element of a reverse iteration (since C++11) |
| c.crend() | Returns a constant reverse iterator for the position after the last element of a reverse iteration (since C++11) |

Sets and multisets do not provide direct element access.

As with all associative container classes, the iterators are bidirectional iterators. You can’t use them in algorithms that are provided only for random-access iterators, such as algorithms for sorting or random shuffling.

From an iterator’s point of view, all elements are considered constant. This is necessary to ensure that you can’t compromise the order of the elements by changing their values. However, as a result, you can’t call any modifying algorithm on the elements of a set or a multiset.

To remove elements (or modify elements) in sets and multisets, you can use only member functions provided by the container.

## Inserting and Removing Elements

|  |  |
| --- | --- |
| Operation | Effect |
| c.insert(val) | Inserts a copy of val and returns the position of the new element and, for sets, whether it succeeded |
| c.insert(pos, val) | Inserts a copy of val and returns the position of the new element (pos is used as a hint pointing to where the insert should start the search) |
| c.insert(beg, end) | Inserts a copy of all elements of the range [beg,end) (returns nothing) |
| c.insert(initlist) | Inserts a copy of all elements in the initializer list initlist (returns nothing; since C++11) |
| c.emplace(args...) | Inserts a copy of an element initialized with args and returns the position of the new element and, for sets, whether it succeeded (since C++11) |
| c.emplace\_hint(pos, args...) | Inserts a copy of an element initialized with args and returns the position of the new element (pos is used as a hint pointing to where the insert should start the search) |
| c.erase(val) | Removes all elements equal to val and returns the number of removed elements |
| c.erase(pos) | Removes the element at iterator position pos and returns the following position (returned nothing before C++11) |
| c.erase(beg, end) | Removes all elements of the range [beg,end) and returns the following position (returned nothing before C++11) |
| c.clear() | Removes all elements (empties the container) |

Inserting and removing is faster if, when working with multiple elements, you use a single call for all elements rather than multiple calls.

For multisets, since C++11 it is guaranteed that insert(), emplace(), and erase() preserve the relative ordering of equivalent elements, and that inserted elements are placed at the end of existing equivalent values.

Return types of the inserting functions insert() and emplace() differ as follows:

**For Sets**

pair<iterator,bool> insert (const value\_type& val);

iterator insert (const\_iterator posHint, const value\_type& val);

template <typename... Args>

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

template <typename... Args>

iterator emplace\_hint (const\_iterator posHint, Args&&... args);

1. The member second of the pair structure returns whether the insertion was successful.
2. The member first of the pair structure returns the position of the newly inserted element or the position of the still existing element.

**For Multisets**

iterator insert (const value\_type& val);

iterator insert (const\_iterator posHint, const value\_type& val);

template <typename... Args>

iterator emplace (Args&&... args);

template <typename... Args>

iterator emplace\_hint (const\_iterator posHint, Args&&... args);

This difference is because multisets allow duplicate whereas sets do not.

In all other cases, the functions return the position of the new element or of the existing element if the set already contains an element with the same value.

Code to process the new or old positions:

auto status = c.insert(value);

if (status.second) { std::cout << value << " inserted as element " }

else { std::cout << value << " already exists as element " }

std::cout << std::distance(c.begin(),status.first) + 1 << std::endl;

In insert function iterator position argument is processed as a hint to optimize performance.

To remove an element that has a certain value, you simply call erase().

If a multiset contains duplicates, you can’t use erase() to remove only the first element of these duplicates.

Instead, you can code as follows:

std::multiset<Elem>::iterator pos;

pos = coll.find(value);

if (pos != coll.end()) {

coll.erase(pos);

}

Use the member function find() instead of the find() algorithm.

# Exception Handling

Sets and multisets are node-based containers, so any failure to construct a node simply leaves the container as it was.

Furthermore, because destructors in general don’t throw, removing a node can’t fail.

However, for multiple-element insert operations, the need to keep elements sorted makes full recovery from throws impractical.

Thus, all single-element insert operations support commit or rollback behavior. That is, they either succeed or have no effect.

In addition, it is guaranteed that all multiple-element delete operations always succeed or have no effect, provided that the comparison criterion does not throw.

If copying/assigning the comparison criterion may throw, swap() may throw.

# END