# Set

Sets are containers that store unique elements following a specific order.  
  
In a set, the value of an element also identifies it (the value is itself the *key*, of type T), and each value must be unique.

The value of the elements in a set cannot be modified once in the container (the elements are always const), but they can be inserted or removed from the container.  
  
Internally, the elements in a set are always sorted following a specific *strict weak ordering* criterion indicated by its internal [comparison object](http://www.cplusplus.com/set::key_comp) (of type Compare).

Sets are typically implemented as *binary search trees*.

## Container properties

***Associative***

Elements in associative containers are referenced by their *key* and not by their absolute position in the container.

***Ordered***

The elements in the container follow a strict order at all times. All inserted elements are given a position in this order.

***Set***

The value of an element is also the *key* used to identify it.

***Unique keys***

No two elements in the container can have equivalent *keys*.

***Allocator-aware***

The container uses an allocator object to dynamically handle its storage needs.

|  |  |  |
| --- | --- | --- |
| **member type** | **definition** | **notes** |
| key\_type | The first template parameter (T) |  |
| value\_type | The first template parameter (T) |  |
| key\_compare | The second template parameter (Compare) | defaults to: [less](http://www.cplusplus.com/less)<key\_type> |
| value\_compare | The second template parameter (Compare) | defaults to: [less](http://www.cplusplus.com/less)<value\_type> |
| allocator\_type | The third template parameter (Alloc) | defaults to: [allocator](http://www.cplusplus.com/allocator)<value\_type> |

## Construct set

Constructs a [set](http://www.cplusplus.com/set) container object, initializing its contents depending on the constructor version used:

***(1) empty container constructors (default constructor)***

Constructs an [empty](http://www.cplusplus.com/set::empty) container, with no elements.

***(2) range constructor***

Constructs a container with as many elements as the range [first,last), with each element *emplace-constructed* from its corresponding element in that range.

***(3) copy constructor (and copying with allocator)***

Constructs a container with a copy of each of the elements in *x*.

***(4) move constructor (and moving with allocator)***

Constructs a container that acquires the elements of *x*.  
If *alloc* is specified and is different from *x*'s allocator, the elements are moved. Otherwise, no elements are constructed (their ownership is directly transferred).  
*x* is left in an unspecified but valid state.

***(5) initializer list constructor***

Constructs a container with a copy of each of the elements in *il*.

// constructing sets

#include <iostream>

#include <set>

bool fncomp (int lhs, int rhs) {return lhs<rhs;}

struct classcomp {

bool operator() (const int& lhs, const int& rhs) const

{return lhs<rhs;}

};

int main ()

{

// ***empty container constructors (default constructor)***

std::set<int> first; // empty container constructor

int myints[]= {10,20,30,40,50};

std::set<int> second (myints,myints+5); // range Constructor

std::set<int> third (second); // a copy of second

std::set<int> fourth (second.begin(), second.end()); // iterator ctor.

std::set<int,classcomp> fifth; // class as Compare

bool(\*fn\_pt)(int,int) = fncomp;

std::set<int,bool(\*)(int,int)> sixth (fn\_pt); // function pointer as Compare

return 0;

}

## std::set::operator=

**Copy container content**

Assigns new contents to the container, replacing its current content.

// assignment operator with sets

#include <iostream>

#include <set>

int main ()

{

int myints[]={ 12,82,37,64,15 };

std::set<int> first (myints,myints+5); // set with 5 ints

std::set<int> second; // empty set

second = first; // now second contains the 5 ints

first = std::set<int>(); // and first is empty

std::cout << "Size of first: " << int (first.size()) << '\n';

std::cout << "Size of second: " << int (second.size()) << '\n';

return 0;

}

Output:

|  |
| --- |
| Size of first: 0  Size of second: 5 |

## Member functions

**Iterators**:

[**begin**](http://www.cplusplus.com/reference/set/set/begin/)

Return iterator to beginning (public member function )

[**end**](http://www.cplusplus.com/reference/set/set/end/)

Return iterator to end (public member function )

[**rbegin**](http://www.cplusplus.com/reference/set/set/rbegin/)

Return reverse iterator to reverse beginning (public member function )

[**rend**](http://www.cplusplus.com/reference/set/set/rend/)

Return reverse iterator to reverse end (public member function )

[**cbegin**](http://www.cplusplus.com/reference/set/set/cbegin/)

Return const\_iterator to beginning (public member function )

[**cend**](http://www.cplusplus.com/reference/set/set/cend/)

Return const\_iterator to end (public member function )

[**crbegin**](http://www.cplusplus.com/reference/set/set/crbegin/)

Return const\_reverse\_iterator to reverse beginning (public member function )

[**crend**](http://www.cplusplus.com/reference/set/set/crend/)

Return const\_reverse\_iterator to reverse end (public member function )

**Capacity**:

[**empty**](http://www.cplusplus.com/reference/set/set/empty/)

Test whether container is empty (public member function )

[**size**](http://www.cplusplus.com/reference/set/set/size/)

Return container size (public member function )

[**max\_size**](http://www.cplusplus.com/reference/set/set/max_size/)

Return maximum size (public member function )

**Modifiers**:

[**insert**](http://www.cplusplus.com/reference/set/set/insert/)

Insert element (public member function )

[**erase**](http://www.cplusplus.com/reference/set/set/erase/)

Erase elements (public member function )

[**swap**](http://www.cplusplus.com/reference/set/set/swap/)

Swap content (public member function )

[**clear**](http://www.cplusplus.com/reference/set/set/clear/)

Clear content (public member function )

[**emplace**](http://www.cplusplus.com/reference/set/set/emplace/)

Construct and insert element (public member function )

[**emplace\_hint**](http://www.cplusplus.com/reference/set/set/emplace_hint/)

Construct and insert element with hint (public member function )

## std::set::insert

**Insert element**

Extends the container by inserting new elements, effectively increasing the container [size](http://www.cplusplus.com/set::size) by the number of elements inserted.  
  
Because elements in a [set](http://www.cplusplus.com/set) are unique, the insertion operation checks whether each inserted element is equivalent to an element already in the container, and if so, the element is not inserted, returning an iterator to this existing element (if the function returns a value).

Internally, [set](http://www.cplusplus.com/set) containers keep all their elements sorted following the criterion specified by its [comparison object](http://www.cplusplus.com/set::key_comp). The elements are always inserted in its respective position following this ordering.

**Return value**

The single element versions (1) return a [pair](http://www.cplusplus.com/pair), with its member

pair::first set to an iterator pointing to either the newly inserted element or to the equivalent element already in the [set](http://www.cplusplus.com/set).

The pair::second element in the [pair](http://www.cplusplus.com/pair) is set to true if a new element was inserted or false if an equivalent element already existed.  
  
The versions with a hint (2) return an iterator pointing to either the newly inserted element or to the element that already had its same value in the [set](http://www.cplusplus.com/set).  
  
Member type iterator is a [bidirectional iterator](http://www.cplusplus.com/BidirectionalIterator) type that points to elements.  
[pair](http://www.cplusplus.com/pair) is a class template declared in [<utility>](http://www.cplusplus.com/%3Cutility%3E)

|  |  |
| --- | --- |
| // set::insert (C++98)  #include <iostream>  #include <set>  int main ()  {  std::set<int> myset;  std::set<int>::iterator it;  std::pair<std::set<int>::iterator,bool> ret;  // set some initial values:  for (int i=1; i<=5; ++i) myset.insert(i\*10); // set: 10 20 30 40 50  ret = myset.insert(20); // no new element inserted  if (ret.second==false) it=ret.first; // "it" now points to element 20  myset.insert (it,25); // max efficiency inserting  myset.insert (it,24); // max efficiency inserting  myset.insert (it,26); // no max efficiency inserting  int myints[]= {5,10,15}; // 10 already in set, not inserted  myset.insert (myints,myints+3);  std::cout << "myset contains:";  for (it=myset.begin(); it!=myset.end(); ++it)  std::cout << ' ' << \*it;  std::cout << '\n';  return 0;  } | [Edit & Run](http://www.cplusplus.com/reference/set/set/insert/) |

Output:

|  |
| --- |
| myset contains: 5 10 15 20 24 25 26 30 40 50 |

## std::[set](http://www.cplusplus.com/reference/set/set/)::clear

**Clear content**

Removes all elements from the [set](http://www.cplusplus.com/set) container (which are destroyed), leaving the container with a [size](http://www.cplusplus.com/set::size) of 0.

|  |  |
| --- | --- |
| // set::clear  #include <iostream>  #include <set>  int main ()  {  std::set<int> myset;  myset.insert (100);  myset.insert (200);  myset.insert (300);  std::cout << "myset contains:";  for (std::set<int>::iterator it=myset.begin(); it!=myset.end(); ++it)  std::cout << ' ' << \*it;  std::cout << '\n';  myset.clear();  myset.insert (1101);  myset.insert (2202);  std::cout << "myset contains:";  for (std::set<int>::iterator it=myset.begin(); it!=myset.end(); ++it)  std::cout << ' ' << \*it;  std::cout << '\n';  return 0;  } | [Edit & Run](http://www.cplusplus.com/reference/set/set/clear/) |

Output:

|  |
| --- |
| myset contains: 100 200 300  myset contains: 1101 2202 |

## std::[set](http://www.cplusplus.com/reference/set/set/)::erase

**Erase elements**

Removes from the [set](http://www.cplusplus.com/set) container either a single element or a range of elements ([first,last)).  
  
This effectively reduces the container [size](http://www.cplusplus.com/set::size) by the number of elements removed, which are destroyed.

**Return value**

For the value-based version (2), the function returns the number of elements erased.

|  |  |
| --- | --- |
| // erasing from set  #include <iostream>  #include <set>  int main ()  {  std::set<int> myset;  std::set<int>::iterator it;  // insert some values:  for (int i=1; i<10; i++) myset.insert(i\*10); // 10 20 30 40 50 60 70 80 90  it = myset.begin();  ++it; // "it" points now to 20  myset.erase (it);  myset.erase (40);  it = myset.find (60);  myset.erase (it, myset.end());  std::cout << "myset contains:";  for (it=myset.begin(); it!=myset.end(); ++it)  std::cout << ' ' << \*it;  std::cout << '\n';  return 0;  } | [Edit & Run](http://www.cplusplus.com/reference/set/set/erase/) |

Output:

|  |
| --- |
| myset contains: 10 30 50 |

**Complexity**

For the first version (erase(position)), amortized constant.  
For the second version (erase(val)), logarithmic in container [size](http://www.cplusplus.com/set::size).  
For the last version (erase(first,last)), linear in the distance between *first* and *last*.

## Member functions

**Observers**:

[**key\_comp**](http://www.cplusplus.com/reference/set/set/key_comp/)

Return comparison object (public member function )

[**value\_comp**](http://www.cplusplus.com/reference/set/set/value_comp/)

Return comparison object (public member function )

**Operations**:

[**find**](http://www.cplusplus.com/reference/set/set/find/)

Get iterator to element (public member function )

[**count**](http://www.cplusplus.com/reference/set/set/count/)

Count elements with a specific value (public member function )

[**lower\_bound**](http://www.cplusplus.com/reference/set/set/lower_bound/)

Return iterator to lower bound (public member function )

[**upper\_bound**](http://www.cplusplus.com/reference/set/set/upper_bound/)

Return iterator to upper bound (public member function )

[**equal\_range**](http://www.cplusplus.com/reference/set/set/equal_range/)

Get range of equal elements (public member function )

**Allocator**:

[**get\_allocator**](http://www.cplusplus.com/reference/set/set/get_allocator/)

Get allocator (public member function )

## std::[set](http://www.cplusplus.com/reference/set/set/)::find

**Get iterator to element**

### Searches the container for an element equivalent to *val* and returns an iterator to it if found, otherwise it returns an iterator to [set::end](http://www.cplusplus.com/set::end). Two elements of a [set](http://www.cplusplus.com/set) are considered equivalent if the container's [comparison object](http://www.cplusplus.com/set::key_comp) returns false reflexively (i.e., no matter the order in which the elements are passed as arguments).

### Parameters

val

Value to be searched for.  
Member type value\_type is the type of the elements in the container, defined in [set](http://www.cplusplus.com/set) as an alias of its first template parameter (T).

**Return value**

An iterator to the element, if *val* is found, or [set::end](http://www.cplusplus.com/set::end) otherwise.  
  
Member types iterator and const\_iterator are [bidirectional iterator](http://www.cplusplus.com/BidirectionalIterator) types pointing to elements.

|  |  |
| --- | --- |
| // set::find  #include <iostream>  #include <set>  int main ()  {  std::set<int> myset;  std::set<int>::iterator it;  // set some initial values:  for (int i=1; i<=5; i++) myset.insert(i\*10); // set: 10 20 30 40 50  it=myset.find(20);  myset.erase (it);  myset.erase (myset.find(40));  std::cout << "myset contains:";  for (it=myset.begin(); it!=myset.end(); ++it)  std::cout << ' ' << \*it;  std::cout << '\n';  return 0;  } | [Edit & Run](http://www.cplusplus.com/reference/set/set/find/) |

Output:

|  |
| --- |
| myset contains: 10 30 50 |

**Complexity**

Logarithmic in [size](http://www.cplusplus.com/set::size).

## std::[set](http://www.cplusplus.com/reference/set/set/)::count

**Count elements with a specific value**

Searches the container for elements equivalent to *val* and returns the number of matches.  
  
Because all elements in a [set](http://www.cplusplus.com/set) container are unique, the function can only return *1* (if the element is found) or zero (otherwise).  
  
Two elements of a [set](http://www.cplusplus.com/set) are considered equivalent if the container's [comparison object](http://www.cplusplus.com/set::key_comp) returns false reflexively (i.e., no matter the order in which the elements are passed as arguments).

**Parameters**

val

Value to search for.  
Member type value\_type is the type of the elements in the container, defined in [set](http://www.cplusplus.com/set) as an alias of its first template parameter (T).

**Return value**

1 if the container contains an element equivalent to *val*, or zero otherwise.  
  
Member type size\_type is an unsigned integral type.

|  |  |
| --- | --- |
| // set::count  #include <iostream>  #include <set>  int main ()  {  std::set<int> myset;  // set some initial values:  for (int i=1; i<5; ++i) myset.insert(i\*3); // set: 3 6 9 12  for (int i=0; i<10; ++i)  {  std::cout << i;  if (myset.count(i)!=0)  std::cout << " is an element of myset.\n";  else  std::cout << " is not an element of myset.\n";  }  return 0;  } | [Edit & Run](http://www.cplusplus.com/reference/set/set/count/) |

Output:

|  |
| --- |
| 0 is not an element of myset.  1 is not an element of myset.  2 is not an element of myset.  3 is an element of myset.  4 is not an element of myset.  5 is not an element of myset.  6 is an element of myset.  7 is not an element of myset.  8 is not an element of myset.  9 is an element of myset. |

**Complexity**

Logarithmic in [size](http://www.cplusplus.com/set::size).

**Iterator validity**

No changes.

# Difference between set, multiset, unordered\_set, unordered\_multiset

## 1. [**Set**](https://www.geeksforgeeks.org/set-in-cpp-stl/)

(i) Stores the values in sorted order.  
(ii) Stores only unique values.  
(iii) Elements can only be inserted or deleted but cannot be modified.  
(iv) We can erase more than 1 element by giving start iterator and end iterator position.  
(v) Traversal using iterators.  
(vi) Sets are implemented as [Binary Search Tree](https://www.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/).

## 2. [**Multiset**](https://www.geeksforgeeks.org/multiset-in-cpp-stl/)

(i) Stores elements in sorted order.  
(ii) It allows storage of multiple elements.  
(iii) We can erase more than 1 element by giving start iterator and end iterator.  
Note:- All other properties similar to set.

3. [**Unordered\_set**](https://www.geeksforgeeks.org/unorderd_set-stl-uses/)  
(i) Elements can be stored in any order. ( no sorted order )  
(ii) Stores only unique values.  
(iii) Hash-table used to store elements.  
(iv) We can erase only the element for which iterator position is given.  
Note:- All other properties similar to set.

4. [**Unordered\_multiset**](https://www.geeksforgeeks.org/unordered_multiset-and-its-uses/)  
(i) Elements can be stored in any order.  
(ii) Duplicate elements can be stored.  
(iii) Hash-table used to store elements.  
(iv) We can erase only the element for which iterator position is given.  
Note:- All other properties similar to set.

**set** is a container that stores **sorted and unique** elements. If **unordered** is added means elements are **not sorted**. If **multiset** is added means **duplicate elements**storage is allowed.

# Difference between set and map

[set](http://www.geeksforgeeks.org/set-in-cpp-stl/) and [map](http://www.geeksforgeeks.org/map-associative-containers-the-c-standard-template-library-stl/) in STL are similar in the sense that they both use [Red Black Tree](https://www.geeksforgeeks.org/red-black-tree-set-1-introduction-2/)(A self balancing [BST](https://www.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/)).

**Note that the time complexities of search, insert and delete are O(Log n).**

|  |  |
| --- | --- |
| **Set** | **Map** |
| set is used to store only keys | map is used to store key-value pairs |
|  |  |
|  |  |

## **Usage**:

1. Consider in the problem of [printing sorted distinct elements](https://www.geeksforgeeks.org/print-sorted-distinct-elements-array-c/), we use set as there is value needed for a key.
2. While if we change the problem to print frequencies of distinct sorted elements, we use map. We need map to store array values as key and frequencies as value.

## **Variations of set and map**:

Set and Map, both stores unique values and sorted values as well.

But If we don’t have such a requirement, we use multiset/multimap and unordered\_set/unoredred\_map.

# Difference between map, multimap, unordered\_map, unordered\_multimap

## Map vs unordered\_map

When it comes to efficiency, there is a huge difference between maps and unordered maps.

|  |  |  |
| --- | --- | --- |
| **Operation** | **Map** | **Unordered Map** |
| Ordering | increasing order  (By Default) | No Ordering |
| Implementation | Self balancing BST  like [Red-Black Tree](https://www.geeksforgeeks.org/red-black-tree-set-1-introduction-2/) | Hash Table |
| search time | Log(n) | O(1) -> Average  O(n) -> Worst Case |
| Insertion time | log(n) + Rebalance | O(1) -> Average  O(n) -> Worst Case |
| Deletion time | log(n) + Rebalance | O(1) -> Average  O(n) -> Worst Case |

**Use std::map when**

* You need ordered data.
* You would have to print/access the data (in sorted order).
* You need predecessor/successor of elements.

|  |
| --- |
| // CPP program to demonstrate use of std::map  #include <bits/stdc++.h>    int main()  {      // Ordered map      std::map<int, int> order;        // Mapping values to keys      order[5] = 10;      order[3] = 5;      order[20] = 100;      order[1] = 1;        // Iterating the map and printing ordered values      for (auto i = order.begin(); i != order.end(); i++) {          std::cout << i->first << " : " << i->second << '\n';      }  } |

Output :

1 : 1

3 : 5

5 : 10

20 : 100

**Use std::unordered\_map when**

* You need to keep count of some data (Example – strings) and no ordering is required.
* You need single element access i.e. no traversal.

## When to choose map instead of unordered\_map

**When you need Low Memory:**

Unordered\_map consumes extra memory for internal hashing, so if you are keeping millions and billions of data inside the map and want to consume less memory then choose std::map instead of std::unordered\_map.

**When you are interested in Ordering too**

As std::map  internally use balanced BST, so all the elements inside it will be in sorted order based on the key. So, if you want all keys to be ordered then go for std::map.

**When you need guaranted Performance**

For searching an element, std::unordered\_map gives the complexity O(1) in best case but O(n) in worst case (if hash implementation is not perfect).

So, if your hash implementation is not good and you have millions and billions of data then go for std::map because it will give you guaranteed O(log N).

## When to choose unordered\_map instead of map

**When you have good hasher and no memory limitation**

Unordered\_map consumes extra memory for internal hashing. But to due to this searching complexity is O(1), if hasher function is good.

|  |
| --- |
| // CPP program to demonstrate use of  // std::unordered\_map  #include <bits/stdc++.h>    int main()  {      // Unordered map      std::unordered\_map<int, int> order;        // Mapping values to keys      order[5] = 10;      order[3] = 5;      order[20] = 100;      order[1] = 1;        // Iterating the map and printing unordered values      for (auto i = order.begin(); i != order.end(); i++) {          std::cout << i->first << " : " << i->second << '\n';      }  } |

Output :

1 : 1

3 : 5

20 : 100

5 : 10

## Map vs Multimap

A std::multimap is equal to a std::map, but your keys are not unique anymore. Therefore you can find a range of items instead of just find one unique item.