map<Key, Data, Compare, Alloc>

Containers

Type

Category: containers

Component type: type

Description

Map is a Sorted Associative Container that associates objects of type Key with objects of type Data. Map is a Pair Associative Container, meaning that its value type is pair<const Key, Data>. It is also a Unique Associative Container, meaning that no two elements have the same key.

Map has the important property that inserting a new element into a map does not invalidate iterators that point to existing elements. Erasing an element from a map also does not invalidate any iterators, except, of course, for iterators that actually point to the element that is being erased.

Example

```
struct ltstr
{
  bool operator()(const char* s1, const char* s2) const
  {
    return strcmp(s1, s2) < 0;
  }
};

int main()
{
  map<const char*, int, ltstr> months;

  months["january"] = 31;
  months["february"] = 28;
  months["march"] = 31;
  months["april"] = 30;
  months["may"] = 31;
  months["june"] = 30;
  months["july"] = 31;
  months["august"] = 31;
  months["september"] = 30;
  months["october"] = 31;
  months["november"] = 30;
  months["december"] = 31;

  cout << "june -> " << months["june"] << endl;
  map<const char*, int, ltstr>::iterator cur  = months.find("june");
  map<const char*, int, ltstr>::iterator prev = cur;
  map<const char*, int, ltstr>::iterator next = cur;
  ++next;
  --prev;
  cout << "Previous (in alphabetical order) is " << (*prev).first << endl;
  cout << "Next (in alphabetical order) is " << (*next).first << endl;
}
```

Definition

Defined in the standard header map, and in the nonstandard backward-compatibility header map.h.

Template parameters

Parameter	Description	Default
`Key`	The map's key type. This is also defined as `map::key_type`.	
`Data`	The map's data type. This is also defined as `map::data_type`.	
`Compare`	The key comparison function, a Strict Weak Ordering whose argument type is `key_type`; it returns `true` if its first argument is less than its second argument, and `false` otherwise. This is also defined as `map::key_compare`.	`less<Key>`
`Alloc`	The `map`'s allocator, used for all internal memory management.	`alloc`

Model of

Unique Sorted Associative Container, Pair Associative Container

Type requirements

- `Data` is Assignable.
- `Compare` is a Strict Weak Ordering whose argument type is `Key`.
- `Alloc` is an Allocator.

Public base classes

None.

Members

Member	Where defined	Description
`key_type`	Associative Container	The `map`'s key type, `Key`.
`data_type`	Pair Associative Container	The type of object associated with the keys.
`value_type`	Pair Associative Container	The type of object, `pair<const key_type, data_type>`, stored in the map.

`key_compare`	Sorted Associative Container	Function object that compares two keys for ordering.
`value_compare`	Sorted Associative Container	Function object that compares two values for ordering.
`pointer`	Container	Pointer to `T`.
`reference`	Container	Reference to `T`
`const_reference`	Container	Const reference to `T`
`size_type`	Container	An unsigned integral type.
`difference_type`	Container	A signed integral type.
`iterator`	Container	Iterator used to iterate through a `map`. [1]
`const_iterator`	Container	Const iterator used to iterate through a `map`.
`reverse_iterator`	Reversible Container	Iterator used to iterate backwards through a `map`. [1]
`const_reverse_iterator`	Reversible Container	Const iterator used to iterate backwards through a `map`.
`iterator begin()`	Container	Returns an `iterator` pointing to the beginning of the `map`.
`iterator end()`	Container	Returns an `iterator` pointing to the end of the `map`.
`const_iterator begin() const`	Container	Returns a `const_iterator` pointing to the beginning of the `map`.
`const_iterator end() const`	Container	Returns a `const_iterator` pointing to the end of the `map`.
`reverse_iterator rbegin()`	Reversible Container	Returns a `reverse_iterator` pointing to the beginning of the reversed map.
`reverse_iterator rend()`	Reversible Container	Returns a `reverse_iterator` pointing to the end of the reversed map.
`const_reverse_iterator rbegin() const`	Reversible Container	Returns a `const_reverse_iterator` pointing to the beginning of the reversed map.
`const_reverse_iterator rend() const`	Reversible Container	Returns a `const_reverse_iterator` pointing to the end of the reversed map.
`size_type size() const`	Container	Returns the size of the `map`.
`size_type max_size() const`	Container	Returns the largest possible size of the `map`.
`bool empty() const`	Container	`true` if the `map`'s size is `0`.
`key_compare key_comp() const`	Sorted	Returns the `key_compare` object used

	Associative Container	by the map.
value_compare value_comp() const	Sorted Associative Container	Returns the value_compare object used by the map.
map()	Container	Creates an empty map.
map(const key_compare& comp)	Sorted Associative Container	Creates an empty map, using comp as the key_compare object.
template <class InputIterator> map(InputIterator f, InputIterator l) [2]	Unique Sorted Associative Container	Creates a map with a copy of a range.
template <class InputIterator> map(InputIterator f, InputIterator l, const key_compare& comp) [2]	Unique Sorted Associative Container	Creates a map with a copy of a range, using comp as the key_compare object.
map(const map&)	Container	The copy constructor.
map& operator=(const map&)	Container	The assignment operator
void swap(map&)	Container	Swaps the contents of two maps.
pair<iterator, bool> insert(const value_type& x)	Unique Associative Container	Inserts x into the map.
iterator insert(iterator pos, const value_type& x)	Unique Sorted Associative Container	Inserts x into the map, using pos as a hint to where it will be inserted.
template <class InputIterator> void insert(InputIterator, InputIterator) [2]	Unique Sorted Associative Container	Inserts a range into the map.
void erase(iterator pos)	Associative Container	Erases the element pointed to by pos.
size_type erase(const key_type& k)	Associative Container	Erases the element whose key is k.
void erase(iterator first, iterator last)	Associative Container	Erases all elements in a range.
void clear()	Associative Container	Erases all of the elements.
iterator find(const key_type& k)	Associative Container	Finds an element whose key is k.
const_iterator find(const key_type& k) const	Associative Container	Finds an element whose key is k.
size_type count(const key_type& k)	Unique Associative Container	Counts the number of elements whose key is k.
iterator lower_bound(const key_type& k)	Sorted Associative	Finds the first element whose key is not less than k.

	Container	
`const_iterator lower_bound(const key_type& k) const`	Sorted Associative Container	Finds the first element whose key is not less than `k`.
`iterator upper_bound(const key_type& k)`	Sorted Associative Container	Finds the first element whose key greater than `k`.
`const_iterator upper_bound(const key_type& k) const`	Sorted Associative Container	Finds the first element whose key greater than `k`.
`pair<iterator, iterator> equal_range(const key_type& k)`	Sorted Associative Container	Finds a range containing all elements whose key is `k`.
`pair<const_iterator, const_iterator> equal_range(const key_type& k) const`	Sorted Associative Container	Finds a range containing all elements whose key is `k`.
`data_type& operator[](const key_type& k)` [3]	`map`	See below.
`bool operator==(const map&, const map&)`	Forward Container	Tests two maps for equality. This is a global function, not a member function.
`bool operator<(const map&, const map&)`	Forward Container	Lexicographical comparison. This is a global function, not a member function.

New members

These members are not defined in the Unique Sorted Associative Container and Pair Associative Container requirements, but are unique to `map`:

Member function	Description
`data_type& operator[](const key_type& k)` [3]	Returns a reference to the object that is associated with a particular key. If the `map` does not already contain such an object, `operator[]` inserts the default object `data_type()`. [3]

Notes

[1] `Map::iterator` is not a mutable iterator, because `map::value_type` is not Assignable. That is, if `i` is of type `map::iterator` and `p` is of type `map::value_type`, then `*i = p` is not a valid expression. However, `map::iterator` isn't a constant iterator either, because it can be used to modify the object that it points to. Using the same notation as above, `(*i).second = p.second` is a valid expression. The same point applies to `map::reverse_iterator`.

[2] This member function relies on *member template* functions, which at present (early 1998) are not supported by all compilers. If your compiler supports member templates, you can call this function with any type of input iterator. If your compiler does not yet support member templates, though, then the arguments must either be of type `const value_type*` or of type `map::const_iterator`.

[3] Since `operator[]` might insert a new element into the `map`, it can't possibly be a `const` member function. Note that the definition of `operator[]` is extremely simple: `m[k]` is equivalent to `(*((m.insert(value_type(k, data_type()))).first)).second`. Strictly speaking, this member function is unnecessary: it exists only for convenience.

See also

Associative Container, Sorted Associative Container, Pair Associative Container, Unique Sorted Associative Container, `set` `multiset`, `multimap`, `hash_set`, `hash_map`, `hash_multiset`, `hash_multimap`,