Bidirectional Map

Generated by Doxygen 1.8.17

1 Main Page

1.1 bidirectional_map

Implementation of a bidirectional associative container in c++. Its goal is to behave similarly to popular stl containers like `std::unordered_map` while providing efficient lookup from key to value as well as from value to key.

1.1.1 Properties

The `bidirectional_map` container contains pairs of values of type K1 and K2.

- Objects in the container are immutable, neither values of type K1 nor values of type K2 can be modified to ensure the integrity of the underlying associative containers
- The container supports the use of different associative containers as base. The default base container is `std::unordered_map` for both forward and inverse lookup. Other tested containers are `std::map` as well as `std::multimap` and `std::unordered_multimap`.
- The mapping from values of K1 to values of K2 is enforced to be injective if the underlying containers for both forward and inverse lookup contain unique keys (like in the default case). This means that for example two pairs (k1, k2) and (k1', k2') can only be inserted at the same time if k1 != k1' **and** k2 != k2'. The use of multimaps as base containers relaxes this constraint.

1.1.2 Doxygen Documentation

- HTML
- PDF

1.1.3 Code Example

An instance of `bidirectional_map` can be created similarly to `std::unordered_map`:

```
#include <string>
#include <unordered_map>
#include "bidirectional_map.hpp"
// empty container
bimap::bidirectional_map<std::string, int> map;
// using initializer list
bimap::bidirectional_map<std::string, int> map1 = {{"Test", 1}, {"Hello", 2}};
// from same container type
bimap::bidirectional_map<std::string, int> map3(map1.begin(), map1.end());
// from different container type
std::unordered_map<std::string, int> values = {{"abc", 1}, {"def", 2}};
bimap::bidirectional_map<std::string, int> map2(values.begin(), values.end());
```

From the items used for initialization only unique ones are inserted (see properties). Further items can be inserted using the `emplace` method

```
bimap::bidirectional_map<std::string, int> map;
map.emplace("NewItem", 17); // constructs new item in place
// no insertion. 17 already exists in the container. Returns iterator to ("NewItem", 17)
auto [iterator, inserted] = map.emplace("AnotherItem", 17);
```

Item lookup:

```
bimap::bidirectional_map<std::string, int> map{{"NewItem", 12}, {"Stuff", 17}};
auto location = map.find("NewItem");
if (location != map.end()) {
    std::cout « location->first « std::endl;
}
if (map.contains("Stuff")) {
    std::cout « map.at("Stuff") « std::endl; // at is only available when using underlying container that
       enforces
                                             // unique keys
}
```

1.1.3.1 Inverse Access

Using the `inverse()` member, inverse lookup and insertion is possible:

```
bimap::bidirectional_map<std::string, int> map;
map.inverse().emplace(123, "one two three"); // inverse insertion
auto invLocation = map.inverse().find(123); // inverse lookup
std::cout « invLocation->first « std::endl; // prints '123'
// inverse of inverse() is again the original
auto location = map.inverse().inverse().find("one two three");
```

`inverse()` returns a reference to `bidirectional_map` where the template types K1 and K2 are reversed. It behaves exactly like the original map except... well the other way around. Even the iterator members are reversed. Copying the `inverse()` container is allowed and will copy the container contents. Moving from `inverse()` is also allowed and behaves as expected.

```
bimap::bidirectional_map<std::string, int> map; // map from std::string -> int
auto inverse = map.inverse(); // independent (copied) container of reversed type (int -> string)
auto &inverseRef = map.inverse(); // inverse access to the same container
```

1.1.3.2 Custom Map Base Container

It is possible to specify a custom map base container for forward lookup as well as for inverse lookup. The default map base type is `std::unordered_map` for forward access as well as for inverse access. Another possible map base type is `std::map`:

```
// only forward access uses the ordered map std::map.
// Inverse access is till provided through std::unordered_map
bimap::bidirectional_map<std::string, int, std::map> map;
// Both forward and inverse access use std::map
bimap::bidirectional_map<std::string, int, std::map, std::map> map1;
```

Another scenario for using a different map base type is when you need to specify for example a custom hash function:

```
struct MyString {...}; // Custom data structure with no default std::hash specialization
struct MyHash {...}; // Custom hash struct
struct MyComparator {...}; // Custom comparator necessary for std::unordered_map
template<typename T, typename U>
using BaseMap = std::unordered_map<T, U, MyHash, MyComparator>;
// for inverse access the default std::unordered_map is sufficient
bimap::bidirectional_map<MyString, int, BaseMap> map;
```

2 Namespace Documentation

2.1 bimap Namespace Reference

namespace containing the bidirectional map class

Namespaces

- impl

  Namespace containing structures and helpers used to implement the bidirectional map. Normally there is no need to use any of its members directly.

Data Structures

- class bidirectional_map

  Bidirectional associative container that supports efficient lookup in both directions.

Functions

- template<typename ForwardKey , typename InverseKey , template< typename ... > typename ForwardMapType = std::unordered_↩ map, template< typename ... > typename InverseMapType = std::unordered_map>
  void swap (bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType > &lhs, bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType > &rhs) noexcept(noexcept(lhs.↩ swap(rhs)))

2.1.1 Detailed Description

namespace containing the bidirectional map class

2.1.2 Function Documentation

2.1.2.1 swap()

```
template<typename ForwardKey , typename InverseKey , template< typename ... > typename ForwardMapType = std::unordered_map, template< typename ... > typename InverseMapType = std::unordered_map>
void bimap::swap (
            bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType > & lhs,
            bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType > & rhs
) [noexcept]
```

See member function bidirectional_map::swap

Parameters

lhs	left hand side
rhs	right hand sode

2.2 bimap::impl Namespace Reference

Namespace containing structures and helpers used to implement the bidirectional map. Normally there is no need to use any of its members directly.

Namespaces

- traits

  namespace containing type traits used in implementation of bidirectional_map

Data Structures

- class AllocOncePointer

  Very simple pointer class that can be used to allocate storage once but can also be used as a non owning pointer.
- class Surrogate

  Non owning pointer to an object. It overloads the equality operators in order to compare the underlying objects instead of the pointer values.

Functions

- template<typename T >
  constexpr auto && get_first (T &&val) noexcept
- template<typename T >
  constexpr void swap (AllocOncePointer< T > &a, AllocOncePointer< T > &b) noexcept

2.2.1 Detailed Description

Namespace containing structures and helpers used to implement the bidirectional map. Normally there is no need to use any of its members directly.

2.2.2 Function Documentation

2.2.2.1 get_first() `template<typename T >`

```
constexpr auto&& bimap::impl::get_first (
            T && val )  [constexpr], [noexcept]
```

Helper function that selects the first member of a tuple

Template Parameters

T	any type

Parameters

val	function argument

Returns

if T is a std::pair, selects the first member. Otherwise, val is forwarded

2.2.2.2 swap() `template<typename T >`

```
constexpr void bimap::impl::swap (
            AllocOncePointer< T > & a,
            AllocOncePointer< T > & b )  [constexpr], [noexcept]
```

See member function AllocOncePointer::swap

Template Parameters

T	type of pointer

Parameters

a	left hand side
b	right hand side

2.3 bimap::impl::traits Namespace Reference

namespace containing type traits used in implementation of bidirectional_map

Data Structures

- struct is_multimap

  type trait that indicates that a given typ is a multimap

Variables

- template<typename T >
  constexpr bool **is_bidirectional_v** = is_bidirectional<T>::value
- template<typename T >
  constexpr bool **is_multimap_v** = is_multimap<T>::value
- template<typename T >
  constexpr bool **nothrow_comparable** = noexcept(std::declval<T>() == std::declval<T>())

2.3.1 Detailed Description

namespace containing type traits used in implementation of bidirectional_map

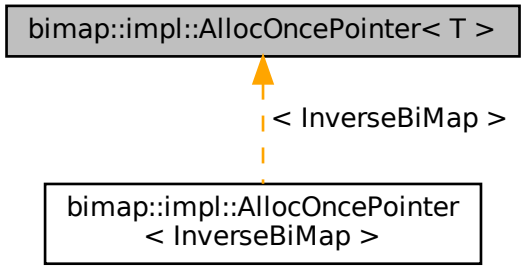
3 Data Structure Documentation

3.1 bimap::impl::AllocOncePointer< T > Class Template Reference

Very simple pointer class that can be used to allocate storage once but can also be used as a non owning pointer.

`#include <bidirectional_map.hpp>`

Inheritance diagram for bimap::impl::AllocOncePointer< T >:



Public Member Functions

- constexpr AllocOncePointer () noexcept
- constexpr AllocOncePointer (T *data) noexcept
- template<typename ... ARGS>
  AllocOncePointer (ARGS &&...args)
- constexpr AllocOncePointer (const AllocOncePointer &other) noexcept
- constexpr void swap (AllocOncePointer &other) noexcept
- constexpr AllocOncePointer (AllocOncePointer &&other) noexcept
- constexpr AllocOncePointer & operator= (AllocOncePointer other) noexcept
- ~AllocOncePointer ()
- constexpr bool isOwner () const noexcept
- constexpr T & operator* () noexcept
- constexpr const T & operator* () const noexcept
- constexpr T * operator-> () noexcept
- constexpr const T * operator-> () const noexcept
- constexpr bool operator== (const AllocOncePointer &other) const noexcept
- constexpr bool operator!= (const T *other) const noexcept

Friends

- constexpr friend bool operator== (const AllocOncePointer &lhs, std::nullptr_t) noexcept
- constexpr friend bool operator== (std::nullptr_t, const AllocOncePointer &rhs) noexcept
- constexpr friend bool operator!= (const AllocOncePointer &lhs, std::nullptr_t) noexcept
- constexpr friend bool operator!= (std::nullptr_t, const AllocOncePointer &rhs) noexcept

3.1.1 Detailed Description

template<typename T>
class bimap::impl::AllocOncePointer< T >

Very simple pointer class that can be used to allocate storage once but can also be used as a non owning pointer.

Unlike shared_ptr, copies of this class are non-owning pointers and unlike weak_ptr, non-owning pointers do not know if the object behind the pointer still exists. The owning pointer deallocates storage at destruction

Template Parameters

T	type of object behind the pointer

3.1.2 Constructor & Destructor Documentation

3.1.2.1 AllocOncePointer() [1/5] template<typename T >

`constexpr bimap::impl::AllocOncePointer< T >::AllocOncePointer ( ) [inline], [constexpr], [noexcept]`

Creates an empty nullptr object

3.1.2.2 AllocOncePointer() [2/5] template<typename T >

```
constexpr bimap::impl::AllocOncePointer< T >::AllocOncePointer (
            T * data )  [inline], [constexpr], [noexcept]
```

Creates a non owning pointer to an existing object

Parameters

data	memory location of object

3.1.2.3 AllocOncePointer() [3/5] template<typename T >

```
template<typename ...  ARGS>
bimap::impl::AllocOncePointer< T >::AllocOncePointer (
            ARGS &&...  args )  [inline], [explicit]
```

Allocates storage and creates an instance of T in place. Becomes owner of the storage

Template Parameters

ARGS	Argument types

Parameters

args	Arguments that are passed to the constructor of T by std::forward

3.1.2.4 AllocOncePointer() [4/5] template<typename T >

```
constexpr bimap::impl::AllocOncePointer< T >::AllocOncePointer (
            const AllocOncePointer< T > & other )  [inline], [constexpr], [noexcept]
```

Copy constructor, creates a non-owning pointer

Parameters

other	source

3.1.2.5 AllocOncePointer() [5/5] template<typename T >

```
constexpr bimap::impl::AllocOncePointer< T >::AllocOncePointer (
            AllocOncePointer< T > && other )  [inline], [constexpr], [noexcept]
```

Move CTor. Takes ownership if other is owning

Parameters

other	source

3.1.2.6 ~AllocOncePointer() template<typename T >

bimap::impl::AllocOncePointer< T >::~AllocOncePointer ( ) [inline]

Destructor. Deallocates memory only when owner

3.1.3 Member Function Documentation

3.1.3.1 isOwner() template<typename T >

constexpr bool bimap::impl::AllocOncePointer< T >::isOwner ( ) const [inline], [constexpr], [noexcept]

Check if pointer is owner

Returns

true if owner

3.1.3.2 operator"!=() template<typename T >

constexpr bool bimap::impl::AllocOncePointer< T >::operator!= (const T * *other*) const [inline], [constexpr], [noexcept]

Parameters

other	right hand side

Returns

true if ∗this is not equal to other

3.1.3.3 operator∗() [1/2] template<typename T >

constexpr const T& bimap::impl::AllocOncePointer< T >::operator* ( ) const [inline], [constexpr], [noexcept]

Dereference operator

Returns

reference to stored data

3.1.3.4 operator*() [2/2] template<typename T >

constexpr T& bimap::impl::AllocOncePointer< T >::operator* ( ) [inline], [constexpr], [noexcept]

Dereference operator

Returns

reference to stored data

3.1.3.5 operator->() [1/2] template<typename T >

constexpr const T* bimap::impl::AllocOncePointer< T >::operator-> ( ) const [inline], [constexpr], [noexcept]

Member access operator

Returns

stored pointer

3.1.3.6 operator->() [2/2] template<typename T >

constexpr T* bimap::impl::AllocOncePointer< T >::operator-> ( ) [inline], [constexpr], [noexcept]

Member access operator

Returns

stored pointer

3.1.3.7 operator=() template<typename T >

constexpr AllocOncePointer& bimap::impl::AllocOncePointer< T >::operator= (AllocOncePointer< T > *other*) [inline], [constexpr], [noexcept]

Assignment operator

Parameters

other	source

Returns

reference to this

3.1.3.8 operator==() template<typename T >

```
constexpr bool bimap::impl::AllocOncePointer< T >::operator== (
            const AllocOncePointer< T > & other ) const  [inline], [constexpr], [noexcept]
```

Equality comparison operator. Compares data pointers

Parameters

other	right hand side

Returns

true if data pointers point to the same object, false otherwise

3.1.3.9 swap() template<typename T >

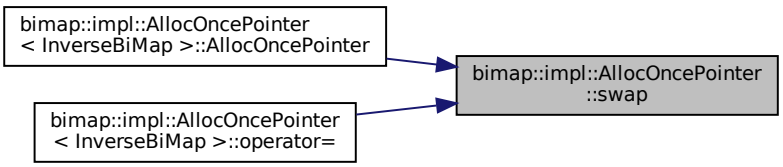
```
constexpr void bimap::impl::AllocOncePointer< T >::swap (
            AllocOncePointer< T > & other )  [inline], [constexpr], [noexcept]
```

Swaps pointer and ownership with other

Parameters

other	swap target

Here is the caller graph for this function:



3.1.4 Friends And Related Function Documentation

3.1.4.1 operator"!= [1/2] template<typename T >

```
constexpr friend bool operator!= (
            const AllocOncePointer< T > & lhs,
            std::nullptr_t  )  [friend]
```

Inequality comparison operator. Compares data pointers

Parameters

lhs	left hand side

Returns

true if data is not nullptr

3.1.4.2 operator"!= [2/2] template<typename T >

```
constexpr friend bool operator!= (
            std::nullptr_t ,
            const AllocOncePointer< T > & rhs )  [friend]
```

Parameters

rhs	right hand side

Returns

true if data is not nullptr

3.1.4.3 operator== [1/2] template<typename T >

```
constexpr friend bool operator== (
            const AllocOncePointer< T > & lhs,
            std::nullptr_t  )  [friend]
```

Parameters

lhs	left hand side

Returns

true if data is not nullptr

3.1.4.4 operator== [2/2] template<typename T >

```
constexpr friend bool operator== (
```

```
std::nullptr_t ,
const AllocOncePointer< T > & rhs )  [friend]
```

Parameters

rhs	right hande side

Returns

true if data is not nullptr

The documentation for this class was generated from the following file:

- bidirectional_map.hpp

3.2 bimap::bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType > Class Template Reference

Bidirectional associative container that supports efficient lookup in both directions.

`#include <bidirectional_map.hpp>`

Data Structures

- class iterator

  bidirectional_map iterator

Public Member Functions

- bidirectional_map ()
- template<typename InputIt >
  bidirectional_map (InputIt start, InputIt end)
- bidirectional_map (std::initializer_list< std::pair< ForwardKey, InverseKey >> init)
- bidirectional_map (const bidirectional_map &other)
- void swap (bidirectional_map &other) noexcept(std::is_nothrow_swappable_v< ForwardMap > &&std::is_↩ nothrow_swappable_v< InverseMap >)
- bidirectional_map (bidirectional_map &&other)
- bidirectional_map & operator= (bidirectional_map other) noexcept(noexcept(std::declval< bidirectional_map >().swap(other)))
- template<typename ... ARGS>
  auto emplace (ARGS &&...args) -> std::pair< iterator, bool >
- auto size () const noexcept(noexcept(std::declval< ForwardMap >().size()))
- bool empty () const noexcept(noexcept(std::declval< ForwardMap >().empty()))
- constexpr auto inverse () noexcept -> InverseBiMap &
- constexpr auto inverse () const noexcept -> const InverseBiMap &
- iterator begin () const noexcept(noexcept(std::declval< ForwardMap >().begin()) &&iterator_ctor_nothrow)
- iterator end () const noexcept(noexcept(std::declval< ForwardMap >().end()) &&iterator_ctor_nothrow)
- iterator find (const ForwardKey &key) const noexcept(noexcept(std::declval< ForwardMap >().find(key)) &&iterator_ctor_nothrow)
- template<REQUIRES_THAT(ForwardMap, std::declval< _T_ >().lower_bound(std::declval< ForwardKey >())) >
  auto lower_bound (const ForwardKey &key) const noexcept(noexcept(std::declval< ForwardMap >().lower↩ _bound(key)) &&iterator_ctor_nothrow) -> iterator

- template<REQUIRES_THAT(ForwardMap, std::declval< _T_ >().upper_bound(std::declval< ForwardKey >())) >
  auto upper_bound (const ForwardKey &key) const noexcept(noexcept(std::declval< ForwardMap >().upper_bound(key)) &&iterator_ctor_nothrow) -> iterator
- auto equal_range (const ForwardKey &key) const noexcept(noexcept(std::declval< ForwardMap >().equal↩ _range(key)) &&iterator_ctor_nothrow) -> std::pair< iterator, iterator >
- iterator erase (iterator pos)
- std::size_t erase (const ForwardKey &key)
- iterator erase (iterator first, iterator last)
- bool operator== (const bidirectional_map &other) const noexcept(impl::traits::nothrow_comparable< ForwardMap > &&impl::traits::nothrow_comparable< InverseMap >)
- bool operator!= (const bidirectional_map &other) const noexcept(noexcept(other==other))
- void clear () noexcept(noexcept(std::declval< ForwardMap >().clear()) &&noexcept(std::declval< Inverse↩ Map >().clear()))
- bool contains (const ForwardKey &key) const noexcept(noexcept(std::declval< bidirectional_map >().find(key)) &&noexcept(std::declval< iterator >() !=std::declval< iterator >()))
- template<bool UniqueKeys = !impl::traits::is_multimap_v<ForwardMap>>
  auto at (const ForwardKey &key) const -> std::enable_if_t< UniqueKeys, const InverseKey & >

Friends

- class **impl::AllocOncePointer< bidirectional_map >**

3.2.1 Detailed Description

template<typename ForwardKey, typename InverseKey, template< typename ... > typename ForwardMapType = std↩ ::unordered_map, template< typename ... > typename InverseMapType = std::unordered_map>
class bimap::bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType >

Bidirectional associative container that supports efficient lookup in both directions.

This class manages two unidirectional maps in order to enable bidirectional lookup. Neither items of type Forward↩ Key nor InverseKey can be modified. The map types for forward and for inverse lookup can be changed. The following map types are supported and have been tested:

- std::unordered_map (default for both lookup directions)
- std::map
- std::unordered_multimap
- std::multimap

Template Parameters

ForwardKey	Type of key used for forward lookup
InverseKey	Type of key used for inverse lookup
ForwardMapType	base map container used for forward lookup. Default is std::unordered_map
InverseMapType	base map container used for inverse lookup. Default is std::unordered_map

Note

when specifying the underlying map types, make sure that the respective types expect two template type arguments. Further arguments have to be deducible or have defaults. Using a custom map type not included in the list should be possible. Make sure that the typical map member functions (like find, emplace, etc) are supported and behave similar to the stl containers. If your map type is a multimap, you have to specialise the type trait impl::traits::is_multimap

3.2.2 Constructor & Destructor Documentation

3.2.2.1 bidirectional_map() [1/5]

```
template<typename ForwardKey , typename InverseKey , template< typename ... > typename ForwardMapType = std::unordered_map, template< typename ... > typename InverseMapType = std::unordered_map>
bimap::bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType >::bidirectional_map ( ) [inline]
```

Creates an empty container

3.2.2.2 bidirectional_map() [2/5]

```
template<typename ForwardKey , typename InverseKey , template< typename ... > typename ForwardMapType = std::unordered_map, template< typename ... > typename InverseMapType = std::unordered_map>
template<typename InputIt >
bimap::bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType >::bidirectional_map (
            InputIt start,
            InputIt end ) [inline]
```

Creates the container from the iterator range [start, end)

Template Parameters

InputIt	Type of iterator

Parameters

start	bein of range (inclusive)
end	end of range (exclusive)

3.2.2.3 bidirectional_map() [3/5]

```
template<typename ForwardKey , typename InverseKey , template< typename ... > typename ForwardMapType = std::unordered_map, template< typename ... > typename InverseMapType = std::unordered_map>
bimap::bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType >::bidirectional_map (
            std::initializer_list< std::pair< ForwardKey, InverseKey >> init ) [inline]
```

Creates the container from the given initializer list

Parameters

init	list of value pairs

3.2.2.4 bidirectional_map() [4/5]

```
template<typename ForwardKey , typename InverseKey , template< typename ... > typename ForwardMapType = std::unordered_map, template< typename ... > typename InverseMapType = std::unordered_map>
bimap::bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType >::bidirectional_map (
            const bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType > & other ) [inline]
```

Copy constructor

Parameters

other	source

3.2.2.5 bidirectional_map() [5/5]

```
template<typename ForwardKey , typename InverseKey , template< typename ... > typename ForwardMapType = std::unordered_map, template< typename ... > typename InverseMapType = std::unordered_map>
bimap::bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType >::bidirectional_map (
            bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType > && other ) [inline]
```

Move constructor. Moves objects from other. If ForwardMapType and InverseMapType support moving, no objects are copied

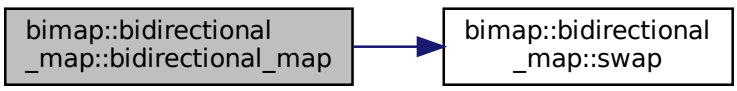
Parameters

other	source

Note

this move CTor may throw exceptions if memory allocation fails

Here is the call graph for this function:



3.2.3 Member Function Documentation

3.2.3.1 at() `template<typename ForwardKey , typename InverseKey , template< typename ... > typename ForwardMapType = std::unordered_map, template< typename ... > typename InverseMapType = std::unordered_map>`

```
template<bool UniqueKeys = !impl::traits::is_multimap_v<ForwardMap>>
auto bimap::bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType >::at (
            const ForwardKey & key ) const -> std::enable_if_t<UniqueKeys, const InverseKey
&>   [inline]
```

Returns the value found by the given key

Parameters

key	key used for lookup

Returns

reference to found value

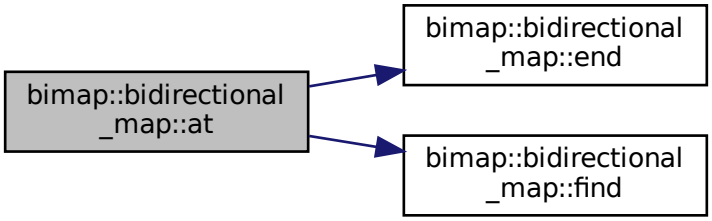
Exceptions

out_of_range	if key does not exist

Note

not available when using multimap as base container

Here is the call graph for this function:



3.2.3.2 begin()

```
template<typename ForwardKey , typename InverseKey , template< typename ... >
typename ForwardMapType = std::unordered_map, template< typename ... > typename InverseMap↩
Type = std::unordered_map>
iterator bimap::bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType >↩
::begin ( ) const  [inline], [noexcept]
```

iterator to first element

Note

Ordering of objects depends on the underlying container specified by ForwardMpaType and InverseMapType. Ordering of forward access may be different from ordering of inverse access

Returns

iterator to first element of forward lookup map

3.2.3.3 clear()

```
template<typename ForwardKey , typename InverseKey , template< typename ... >
typename ForwardMapType = std::unordered_map, template< typename ... > typename InverseMap↩
Type = std::unordered_map>
void bimap::bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType >::clear
( )  [inline], [noexcept]
```

Erases all elements from the container

3.2.3.4 contains()

```
template<typename ForwardKey , typename InverseKey , template< typename ...
> typename ForwardMapType = std::unordered_map, template< typename ... > typename Inverse↩
MapType = std::unordered_map>
bool bimap::bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType >↩
::contains (
            const ForwardKey & key ) const  [inline], [noexcept]
```

Check if a certain key can be found

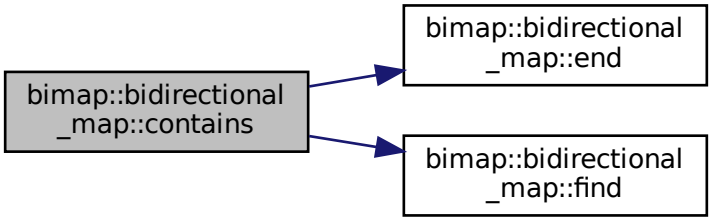
Parameters

key	key used for lookup

Returns

true if key can be found, false otherwise

Here is the call graph for this function:



3.2.3.5 emplace()

```
template<typename ForwardKey , typename InverseKey , template< typename ...
> typename ForwardMapType = std::unordered_map, template< typename ...  > typename Inverse↩
MapType = std::unordered_map>
template<typename ...  ARGS>
auto bimap::bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType >↩
::emplace (
             ARGS &&...  args ) -> std::pair<iterator, bool>   [inline]
```

Constructs elements in place. If a pair of values with same ForwardKey or same InverseKey already exists and the corresponding container requires unique keys, then no insertion happens. For example, if std::multiset is used for forward lookup and the map contains the following pair :(a, b) then inserting (a, b') is possible whereas (a', b) will not be inserted since the inverse lookup is carried out by std::unordered_map

Template Parameters

ARGS	argument types

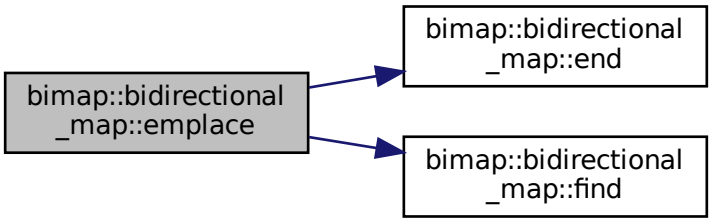
Parameters

args	arguments used to construct elements

Returns

std::pair(iterator to inserted element or already existing element, bool whether insertion happened)

Here is the call graph for this function:



3.2.3.6 empty()

```
template<typename ForwardKey , typename InverseKey , template< typename ... > typename ForwardMapType = std::unordered_map, template< typename ... > typename InverseMapType = std::unordered_map>
bool bimap::bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType >::empty ( ) const [inline], [noexcept]
```

Whether container is empty

Returns

true if container is empty

3.2.3.7 end()

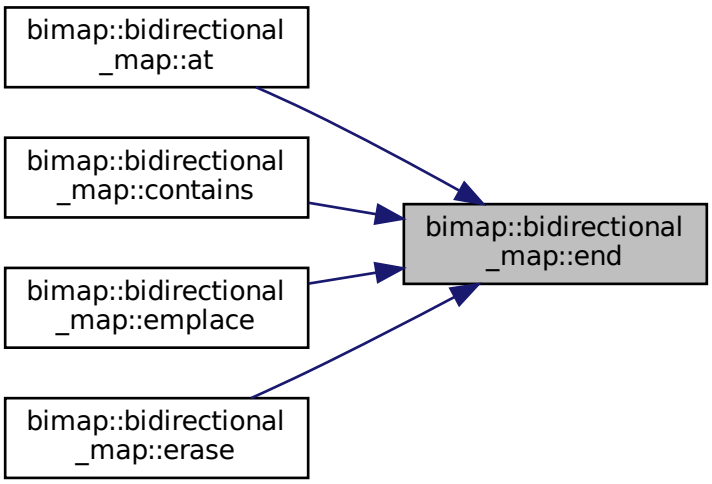
```
template<typename ForwardKey , typename InverseKey , template< typename ... > typename ForwardMapType = std::unordered_map, template< typename ... > typename InverseMapType = std::unordered_map>
iterator bimap::bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType >::end ( ) const [inline], [noexcept]
```

iterator to the past the end element. This iterator does not point to anything. Access results in undefined behaviour

Returns

iterator to past the end element of forward lookup map

Here is the caller graph for this function:



3.2.3.8 equal_range()

```
template<typename ForwardKey , typename InverseKey , template< typename
...  > typename ForwardMapType = std::unordered_map, template< typename ...  > typename Inverse↩
MapType = std::unordered_map>
auto bimap::bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType >↩
::equal_range (
            const ForwardKey & key ) const -> std::pair<iterator, iterator>   [inline],
[noexcept]
```

Calls equal_range on the underlying container. For more information see documentation of the respective container type.

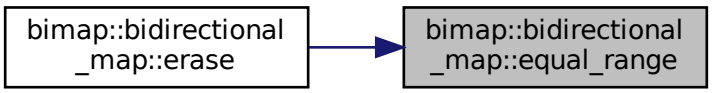
Parameters

key	Key used for lookup

Returns

iterator range containing equal elements

Here is the caller graph for this function:



3.2.3.9 erase() [1/3]

```
template<typename ForwardKey , typename InverseKey , template< typename
...  > typename ForwardMapType = std::unordered_map, template< typename ...  > typename Inverse↩
MapType = std::unordered_map>
std::size_t bimap::bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType
>::erase (
            const ForwardKey & key )  [inline]
```

Erases all elements with forward key equivalent to key.

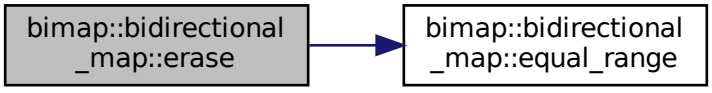
Parameters

key	key used for lookup

Returns

number of erased elements

Here is the call graph for this function:



3.2.3.10 erase() [2/3]

```
template<typename ForwardKey , typename InverseKey , template< typename ... > typename ForwardMapType = std::unordered_map, template< typename ... > typename InverseMapType = std::unordered_map>
iterator bimap::bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType >::erase (
            iterator first,
            iterator last ) [inline]
```

Erases all elements in the range [first, last) which must be a valid range in ∗this

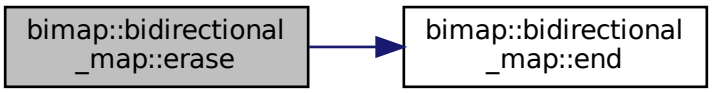
Parameters

first	start of the range (inclusive)
last	end of the range (exclusive)

Returns

iterator following the last removed element

Here is the call graph for this function:



3.2.3.11 erase() [3/3]

```
template<typename ForwardKey , typename InverseKey , template< typename ... > typename ForwardMapType = std::unordered_map, template< typename ... > typename InverseMapType = std::unordered_map>
iterator bimap::bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType >::erase (
            iterator pos ) [inline]
```

Erases the element at position pos

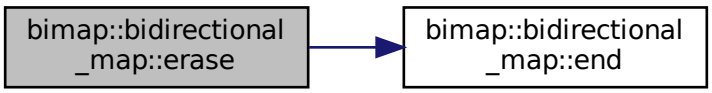
Parameters

pos	iterator to the element to remove. if pos == end(), this method does nothing

Returns

iterator pointing to the next element in the container

Here is the call graph for this function:



3.2.3.12 find()

```
template<typename ForwardKey , typename InverseKey , template< typename ... > typename ForwardMapType = std::unordered_map, template< typename ... > typename InverseMapType = std::unordered_map>
iterator bimap::bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType >::find (
            const ForwardKey & key ) const  [inline], [noexcept]
```

Finds an element with forward key equivalent to key

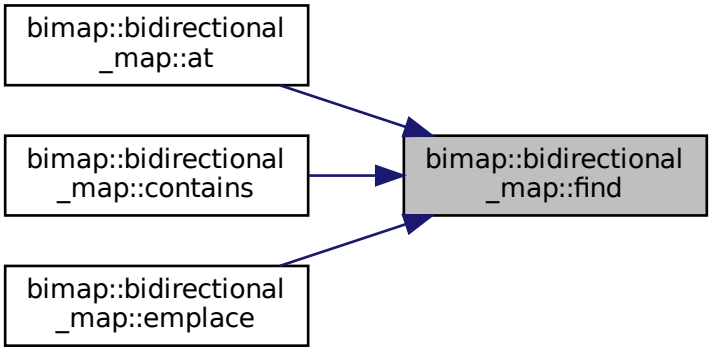
Parameters

key	key used for lookup

Returns

iterator to an element with forward key equivalent to key. If no such element is found, past-the-end (see end()) iterator is returned.

Here is the caller graph for this function:



3.2.3.13 inverse() [1/2]

```
template<typename ForwardKey , typename InverseKey , template< typename
...  > typename ForwardMapType = std::unordered_map, template< typename ...  > typename Inverse↩
MapType = std::unordered_map>
constexpr auto bimap::bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMap↩
Type >::inverse ( ) const -> const InverseBiMap &   [inline], [constexpr], [noexcept]
```

Readonly access to the inverted map for reverse lookup

Returns

const reference to inverted map

3.2.3.14 inverse() [2/2]

```
template<typename ForwardKey , typename InverseKey , template< typename
...  > typename ForwardMapType = std::unordered_map, template< typename ...  > typename Inverse↩
MapType = std::unordered_map>
constexpr auto bimap::bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMap↩
Type >::inverse ( ) -> InverseBiMap &   [inline], [constexpr], [noexcept]
```

Access to the inverted map for reverse lookup or insertion

Returns

Reference to inverted map

3.2.3.15 lower_bound()

```
template<typename ForwardKey , typename InverseKey , template< typename ... > typename ForwardMapType = std::unordered_map, template< typename ... > typename InverseMapType = std::unordered_map>
template<REQUIRES_THAT(ForwardMap, std::declval< _T_ >().lower_bound(std::declval< ForwardKey >())) >
auto bimap::bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType >::lower_bound (
            const ForwardKey & key ) const -> iterator   [inline], [noexcept]
```

Calls lower_bound on the underlying container. For more information see documentation of the respective container type. Only available when using sorted containers like std::map

Parameters

key	Key used for lookup

Returns

lower bound iterator

3.2.3.16 operator"!=()

```
template<typename ForwardKey , typename InverseKey , template< typename ... > typename ForwardMapType = std::unordered_map, template< typename ... > typename InverseMapType = std::unordered_map>
bool bimap::bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType >::operator!= (
            const bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType > & other ) const  [inline], [noexcept]
```

Compares container by elements, see operator==

Parameters

other	right hand side

Returns

true if *this != other

3.2.3.17 operator=()

```
template<typename ForwardKey , typename InverseKey , template< typename ... > typename ForwardMapType = std::unordered_map, template< typename ... > typename InverseMapType = std::unordered_map>
bidirectional_map& bimap::bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType >::operator= (
            bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType > other ) [inline], [noexcept]
```

Assignment operator

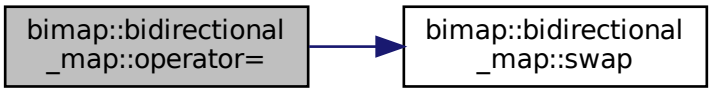
Parameters

other	source

Returns

reference to ∗this

Here is the call graph for this function:



3.2.3.18 operator==()

```
template<typename ForwardKey , typename InverseKey , template< typename
...  > typename ForwardMapType = std::unordered_map, template< typename ...  > typename Inverse↩
MapType = std::unordered_map>
bool bimap::bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType >↩
::operator== (
            const bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType >
& other ) const  [inline], [noexcept]
```

Compares underlying containers

Parameters

other	right hand side

Returns

true if both forward mapping and inverse mapping are equivalent

Note

for more details see documentation of the used underlying containers. If the default containers are used, the underlying std::unordered_maps are compared

3.2.3.19 size()

```
template<typename ForwardKey , typename InverseKey , template< typename ... > typename ForwardMapType = std::unordered_map, template< typename ... > typename InverseMapType = std::unordered_map>
auto bimap::bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType >::size ( ) const [inline], [noexcept]
```

Number of contained elements

Returns

Number of contained elements

3.2.3.20 swap()

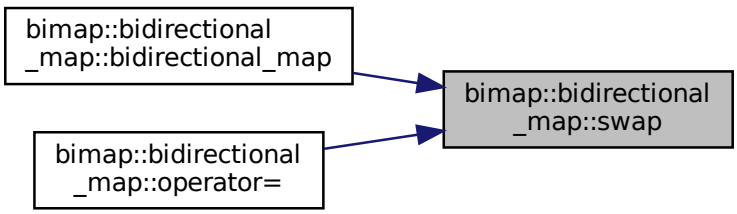
```
template<typename ForwardKey , typename InverseKey , template< typename ... > typename ForwardMapType = std::unordered_map, template< typename ... > typename InverseMapType = std::unordered_map>
void bimap::bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType >::swap (
            bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType > & other ) [inline], [noexcept]
```

Swaps the content of the containers. If ForwardMapType and InverseMapType support moving, no objects are copied

Parameters

other	swap target

Here is the caller graph for this function:



3.2.3.21 upper_bound()

```
template<typename ForwardKey , typename InverseKey , template< typename ... > typename ForwardMapType = std::unordered_map, template< typename ... > typename InverseMapType = std::unordered_map>
template<REQUIRES_THAT(ForwardMap, std::declval< _T_ >().upper_bound(std::declval< ForwardKey >())) >
```

```
auto bimap::bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType >↩
::upper_bound (
            const ForwardKey & key ) const -> iterator   [inline], [noexcept]
```

Calls upper_bound on the underlying container. For more information see documentation of the respective container type. Only available when using sorted containers like std::map

Parameters

key	Key used for lookup

Returns

upper bound iterator

The documentation for this class was generated from the following file:

- bidirectional_map.hpp

3.3 bimap::impl::traits::is_multimap< T > Struct Template Reference

type trait that indicates that a given typ is a multimap

```
#include <bidirectional_map.hpp>
```

Static Public Attributes

- static constexpr bool **value** = false

3.3.1 Detailed Description

template<typename T>
struct bimap::impl::traits::is_multimap< T >

type trait that indicates that a given typ is a multimap

If you want to use a custom multimap type, specialize this trait for said type. Example for a type called MyMulti↩Map

```
template<typename Key, typename Val, typename Stuff>
struct bimap::impl::traits::is_multimap<MyMultiMap<Key, Val, Stuff» : std::true_type {};
```

The documentation for this struct was generated from the following file:

- bidirectional_map.hpp

3.4 bimap::bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType >::iterator Class Reference

bidirectional_map iterator

```
#include <bidirectional_map.hpp>
```

Public Types

- using **value_type** = std::pair< const ForwardKey &, const InverseKey & >
- using **reference** = value_type
- using **pointer** = std::add_pointer_t< std::add_const_t< value_type > >
- using **diference_type** = typename std::iterator_traits< IteratorType >::difference_type
- using **iterator_category** = typename std::iterator_traits< IteratorType >::iterator_category

Public Member Functions

- constexpr iterator (const IteratorType &it) noexcept(copy_constructable)
- constexpr iterator (const iterator &other) noexcept(std::is_constructible_v< iterator, IteratorType >)
- constexpr iterator (iterator &&other) noexcept(std::is_constructible_v< iterator, IteratorType >)
- constexpr iterator & operator= (iterator other) noexcept(copy_assignable)
- constexpr iterator & operator++ () noexcept(noexcept(++std::declval< IteratorType >()))
- constexpr iterator operator++ (int) noexcept(std::is_nothrow_copy_constructible_v< iterator > &&noexcept(++std::declval< iterator >()))
- constexpr bool operator== (const iterator &other) const noexcept(impl::traits::nothrow_comparable< IteratorType >)
- constexpr bool operator!= (const iterator &other) const noexcept(noexcept(other==other))
- constexpr reference operator* () const
- constexpr pointer operator-> () const

bidirectional iterators

the following operators are only available if all underlying iterators support bidirectional access

- template<bool IsBidirectional = impl::traits::is_bidirectional_v<IteratorType>>
  constexpr auto operator-- () noexcept(noexcept(--std::declval< IteratorType >())) -> std::enable_if_t< IsBidirectional, iterator & >
- template<bool IsBidirectional = impl::traits::is_bidirectional_v<IteratorType>>
  constexpr auto operator-- (int) noexcept(std::is_nothrow_copy_constructible_v< iterator > &&noexcept(--std::declval< iterator >())) -> std::enable_if_t< IsBidirectional, iterator >

Friends

- class **bidirectional_map**

3.4.1 Detailed Description

template<typename ForwardKey, typename InverseKey, template< typename ... > typename ForwardMapType = std::unordered_map, template< typename ... > typename InverseMapType = std::unordered_map>
class bimap::bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType >::iterator

bidirectional_map iterator

3.4.2 Constructor & Destructor Documentation

3.4.2.1 iterator() [1/3]

```
template<typename ForwardKey , typename InverseKey , template< typename ... > typename ForwardMapType = std::unordered_map, template< typename ... > typename InverseMapType = std::unordered_map>
constexpr bimap::bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType >::iterator::iterator (
            const IteratorType & it )  [inline], [explicit], [constexpr], [noexcept]
```

CTor

Parameters

it	iterator to underlying map element

3.4.2.2 iterator() [2/3]

```
template<typename ForwardKey , typename InverseKey , template< typename ... > typename ForwardMapType = std::unordered_map, template< typename ... > typename InverseMapType = std::unordered_map>
constexpr bimap::bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType >::iterator::iterator (
            const iterator & other )  [inline], [constexpr], [noexcept]
```

Copy ctor

Parameters

other	source

3.4.2.3 iterator() [3/3]

```
template<typename ForwardKey , typename InverseKey , template< typename ... > typename ForwardMapType = std::unordered_map, template< typename ... > typename InverseMapType = std::unordered_map>
constexpr bimap::bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType >::iterator::iterator (
            iterator && other )  [inline], [constexpr], [noexcept]
```

Move CTor

Parameters

other	source

3.4.3 Member Function Documentation

3.4.3.1 operator"!=()

```
template<typename ForwardKey , typename InverseKey , template< typename ... > typename ForwardMapType = std::unordered_map, template< typename ... > typename InverseMapType = std::unordered_map>
constexpr bool bimap::bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType >::iterator::operator!= (
            const iterator & other ) const  [inline], [constexpr], [noexcept]
```

Inequality operator. Compares underlying map iterators

Parameters

other	right hand side

Returns

true if ∗this != other

3.4.3.2 operator∗() `template<typename ForwardKey , typename InverseKey , template< typename ... > typename ForwardMapType = std::unordered_map, template< typename ... > typename InverseMapType = std::unordered_map>`

`constexpr reference bimap::bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType >::iterator::operator* ( ) const [inline], [constexpr]`

Returns a pair of reference to container elements

Returns

std::pair of references to map elements

Note

when using structured bindings, map elements are captured by const reference

```
bidirectional_map<int, char> map{{1, 'a'}};
auto begin = map.begin();
auto [num, c] = *begin; // num and c are const references, no additional reference binding is necessary
num = 3; // error
```

3.4.3.3 operator++() [1/2] `template<typename ForwardKey , typename InverseKey , template< typename ... > typename ForwardMapType = std::unordered_map, template< typename ... > typename InverseMapType = std::unordered_map>`

`constexpr iterator& bimap::bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType >::iterator::operator++ ( ) [inline], [constexpr], [noexcept]`

Increments underlying iterator by one

Returns

reference to this

3.4.3.4 operator++() [2/2]

```
template<typename ForwardKey , typename InverseKey , template< typename ... > typename ForwardMapType = std::unordered_map, template< typename ... > typename InverseMapType = std::unordered_map>
constexpr iterator bimap::bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType >::iterator::operator++ (
            int  )  [inline], [constexpr], [noexcept]
```

Post increment. Increments underlying iterator by one

Returns

instance of iterator

3.4.3.5 operator--() [1/2]

```
template<typename ForwardKey , typename InverseKey , template< typename ... > typename ForwardMapType = std::unordered_map, template< typename ... > typename InverseMapType = std::unordered_map>
template<bool IsBidirectional = impl::traits::is_bidirectional_v<IteratorType>>
constexpr auto bimap::bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType >::iterator::operator-- ( ) -> std::enable_if_t<IsBidirectional, iterator &>  [inline], [constexpr], [noexcept]
```

Decrements underlying iterator by one. Only available if base iterator supports bidirectional iteration

Template Parameters

IsBidirectional	SFINAE guard. Do not specify

Returns

reference to this

3.4.3.6 operator--() [2/2]

```
template<typename ForwardKey , typename InverseKey , template< typename ... > typename ForwardMapType = std::unordered_map, template< typename ... > typename InverseMapType = std::unordered_map>
template<bool IsBidirectional = impl::traits::is_bidirectional_v<IteratorType>>
constexpr auto bimap::bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType >::iterator::operator-- (
            int  ) -> std::enable_if_t<IsBidirectional, iterator>  [inline], [constexpr], [noexcept]
```

Post decrement. Only available if base iterator supports bidirectional iteration

Template Parameters

IsBidirectional	SFINAE guard. Do not specify

Returns

instance of iterator

3.4.3.7 operator->() template<typename ForwardKey , typename InverseKey , template< typename ... > typename ForwardMapType = std::unordered_map, template< typename ... > typename InverseMapType = std::unordered_map>
constexpr pointer bimap::bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType >::iterator::operator-> ( ) const [inline], [constexpr]

Member access operator

Returns

pointer to reference pair

3.4.3.8 operator=() template<typename ForwardKey , typename InverseKey , template< typename ... > typename ForwardMapType = std::unordered_map, template< typename ... > typename InverseMapType = std::unordered_map>
constexpr iterator& bimap::bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType >::iterator::operator= (
iterator *other*) [inline], [constexpr], [noexcept]

Assignment operator

Parameters

other	source

Returns

reference to this

3.4.3.9 operator==() template<typename ForwardKey , typename InverseKey , template< typename ... > typename ForwardMapType = std::unordered_map, template< typename ... > typename InverseMapType = std::unordered_map>
constexpr bool bimap::bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType >::iterator::operator== (
const iterator & *other*) const [inline], [constexpr], [noexcept]

Equality operator. Compares underlying map iterators

Parameters

other	right hand side

Returns

true if underlying iterators are equal

The documentation for this class was generated from the following file:

- bidirectional_map.hpp

3.5 bimap::impl::Surrogate< T > Class Template Reference

Non owning pointer to an object. It overloads the equality operators in order to compare the underlying objects instead of the pointer values.

`#include <bidirectional_map.hpp>`

Public Member Functions

- constexpr Surrogate (T *data) noexcept
- constexpr bool operator== (Surrogate other) const noexcept(traits::nothrow_comparable< T >)
- constexpr bool operator!= (Surrogate other) const noexcept(noexcept(other==other))
- constexpr T & operator* () noexcept
- constexpr const T & operator* () const noexcept
- constexpr T * operator-> () noexcept
- constexpr const T * operator-> () const noexcept
- constexpr T * get () noexcept
- constexpr const T * get () const noexcept

3.5.1 Detailed Description

template<typename T>
class bimap::impl::Surrogate< T >

Non owning pointer to an object. It overloads the equality operators in order to compare the underlying objects instead of the pointer values.

Template Parameters

T	type of object behind the pointer

3.5.2 Constructor & Destructor Documentation

3.5.2.1 Surrogate() `template<typename T >`

```
constexpr bimap::impl::Surrogate< T >::Surrogate (
            T * data )  [inline], [constexpr], [noexcept]
```

CTor. Stores pointer to data.

Parameters

data	memory location of data

Note

Instances of this type are always non-owning

3.5.3 Member Function Documentation

3.5.3.1 get() [1/2] template<typename T >

```
constexpr const T* bimap::impl::Surrogate< T >::get ( ) const  [inline], [constexpr], [noexcept]
```

Getter for stored pointer

Returns

raw pointer to data

3.5.3.2 get() [2/2] template<typename T >

```
constexpr T* bimap::impl::Surrogate< T >::get ( )  [inline], [constexpr], [noexcept]
```

Getter for stored pointer

Returns

raw pointer to data

3.5.3.3 operator"!=() template<typename T >

```
constexpr bool bimap::impl::Surrogate< T >::operator!= (
            Surrogate< T > other ) const  [inline], [constexpr], [noexcept]
```

Compares objects behind the pointer

Parameters

other	right hand side

Returns

true if ∗this is not equal to other

3.5.3.4 operator*() [1/2] template<typename T >

```
constexpr const T& bimap::impl::Surrogate< T >::operator* ( ) const  [inline], [constexpr],
[noexcept]
```

Dereference operator

Returns

reference to stored data

3.5.3.5 operator*() [2/2] template<typename T >

```
constexpr T& bimap::impl::Surrogate< T >::operator* ( )  [inline], [constexpr], [noexcept]
```

Dereference operator

Returns

reference to stored data

3.5.3.6 operator->() [1/2] template<typename T >

```
constexpr const T* bimap::impl::Surrogate< T >::operator-> ( ) const  [inline], [constexpr],
[noexcept]
```

Member access operator

Returns

stored pointer

3.5.3.7 operator->() [2/2] template<typename T >

```
constexpr T* bimap::impl::Surrogate< T >::operator-> ( )  [inline], [constexpr], [noexcept]
```

Member access operator

Returns

stored pointer

3.5.3.8 operator==() template<typename T >

```
constexpr bool bimap::impl::Surrogate< T >::operator== (
            Surrogate< T > other ) const  [inline], [constexpr], [noexcept]
```

Compares objects behind the pointer

Parameters

other	right hand side

Returns

true if underlying objects compare equal

Note

unlike for example std::shared_ptr, the actual objects behind the pointers are compared, not the pointer values themselves. This requires that both left and right hand side point to valid memory locations

The documentation for this class was generated from the following file:

- bidirectional_map.hpp

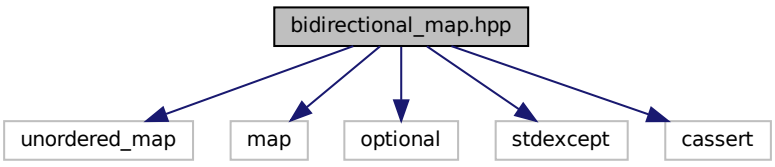
4 File Documentation

4.1 bidirectional_map.hpp File Reference

This file contains the class definition of a bidirectional associative container that can be used for efficient lookup in both directions. Its contents are immutable to ensure the integrity of the underlying map map containers.

```
#include <unordered_map>
#include <map>
#include <optional>
#include <stdexcept>
#include <cassert>
```

Include dependency graph for bidirectional_map.hpp:



Data Structures

- struct bimap::impl::traits::is_multimap< T >

  type trait that indicates that a given typ is a multimap
- class bimap::impl::AllocOncePointer< T >

  Very simple pointer class that can be used to allocate storage once but can also be used as a non owning pointer.
- class bimap::impl::Surrogate< T >

  Non owning pointer to an object. It overloads the equality operators in order to compare the underlying objects instead of the pointer values.
- class bimap::bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType >

  Bidirectional associative container that supports efficient lookup in both directions.
- class bimap::bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType >::iterator

  bidirectional_map iterator

Namespaces

- bimap

  namespace containing the bidirectional map class
- bimap::impl

  Namespace containing structures and helpers used to implement the bidirectional map. Normally there is no need to use any of its members directly.
- bimap::impl::traits

  namespace containing type traits used in implementation of bidirectional_map

Macros

- #define **REQUIRES_THAT**(TYPENAME, EXPRESSION) typename _T_ = TYPENAME, typename = std::void_t<decltype(EXPRESSION)>

Functions

- template<typename T >
  constexpr auto && bimap::impl::get_first (T &&val) noexcept
- template<typename T >
  constexpr void bimap::impl::swap (AllocOncePointer< T > &a, AllocOncePointer< T > &b) noexcept
- template<typename ForwardKey , typename InverseKey , template< typename ... > typename ForwardMapType = std::unordered_map, template< typename ... > typename InverseMapType = std::unordered_map>
  void bimap::swap (bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType > &lhs, bidirectional_map< ForwardKey, InverseKey, ForwardMapType, InverseMapType > &rhs) noexcept(noexcept(lhs.swap(rhs)))

Variables

- template<typename T >
  constexpr bool **bimap::impl::traits::is_bidirectional_v** = is_bidirectional<T>::value
- template<typename T >
  constexpr bool **bimap::impl::traits::is_multimap_v** = is_multimap<T>::value
- template<typename T >
  constexpr bool **bimap::impl::traits::nothrow_comparable** = noexcept(std::declval<T>() == std::declval<T>())

4.1.1 Detailed Description

This file contains the class definition of a bidirectional associative container that can be used for efficient lookup in both directions. Its contents are immutable to ensure the integrity of the underlying map map containers.

Author

Tim Luchterhand

Date

2021-06-16

Index