Boost.Bimap

Matias Capeletto

Table of Contents

Preface

Description

Boost.Bimap is a bidirectional maps library for C++. With Boost.Bimap you can create associative containers in which both types can be used as key. A `bimap<X,Y>` can be thought of as a combination of a `std::map<X,Y>` and a `std::map<Y,X>`. The learning curve of bimap is almost flat if you know how to use standard containers. A great deal of effort has been put into mapping the naming scheme of the STL in Boost.Bimap. The library is designed to match the common STL containers.

Influences and Related Work

The design of Boost.Bimap interface follows the standard template library. It has been strongly influenced by Joaquin Lopez Muñoz's Boost.MultiIndex library (the heart of bimaps) and codeproject::bimap library.

Introduction

How to use this document

This documentation contains a large amount of information. Whereas it may be worth reading it all, this documentation is intended for programmers with various motives:

I have to finished this today, I just want a bidirectional map!	If your boss will kill you if the project is not finished by the end of the day, just read the One-minute tutorial. If you have a background in STL, you can be testing a bimap within ten minutes.
I am a serious programmer and want to learn Boost.Bimap	Boost.Bimap has a lot to offer if you are prepared to spend some time reading this documentation. You will need to read The tutorial and skim through some of the Examples. The best way to read this documentation is in the order given here. Just click on the arrow at the right bottom corner as you finish each page. You may skip the reference section, and return to it later to look up a function signature or to find a specific metafunction.
I just love C++, I want to see the inner workings of Boost.Bimap.	If you are a library developer, this documentation is the best place to learn how Boost.Bimap is implemented. It is strongly recommended that you first learn to use the library as if you were the second type of programmer above. This library was developed in the Google SoC 2006, and the mentor and student generated a great deal of documentation in the building process. The rationale section is very large and contains a lot of information. There is a history section for those who might find it useful. Finally, in the reference section, each entity of the library is documented and its source code is presented.

> **Note**
>
> If anything in the documentation is unclear, please email me at *matias {dot} capeletto {at} gmail {dot} com*, telling me which of the three types of programmer above you are and which section needs improvement. Please use the following notation for the subject: *[boost][bimap] Your problem* as this will help me to identify it more easily. If appropriate, I will act on your advice to improve the documentation. Thanks and enjoy!

> **Important**
>
> If you should find a bug or would like to see an additional feature in the library, please use the standard Boost methods of dealing with this kind of issue rather than emailing me directly. Boost has a very good system to track bugs and features requests, and using it is the best way of dealing with them as soon as possible.

Navigation

Used in combination with the configured browser key (usually Alt), the following keys act as handy shortcuts for common navigation tasks.

- **General**

  - `p` - Previous page

  - `n` - Next page

  - `h` - home

  - `u` - Up

- **Main TOC**

  - `i` - Introduction

- **o** - One minute tutorial
- **t** - The tutorial
- **b** - Bimap and Boost
- **r** - Reference
- **c** - Compiler specifics
- **v** - Performance
- **e** - Examples
- **s** - Test Suite
- **f** - Future work
- **m** - Release notes
- **w** - Rationale
- **y** - History
- **a** - Acknowledgements

One minute tutorial

What is a bimap?

A Bimap is a data structure that represents bidirectional relations between elements of two collections. The container is designed to work as two opposed STL maps. A bimap between a collection X and a collection Y can be viewed as a map from X to Y (this view will be called the *left map view*) or as a map from Y to X (known as the *right map view*). Additionally, the bimap can also be viewed as a set of relations between X and Y (named the *collection of relations view*).

The following code creates an empty bimap container:

```
typedef bimap<X,Y> bm_type;
bm_type bm;
```

Given this code, the following is the complete description of the resulting bimap. [1]

- `bm.left` is signature-compatible with `std::map<X,Y>`

- `bm.right` is signature-compatible with `std::map<Y,X>`

- `bm` is signature-compatible with `std::set< relation<X,Y> >`
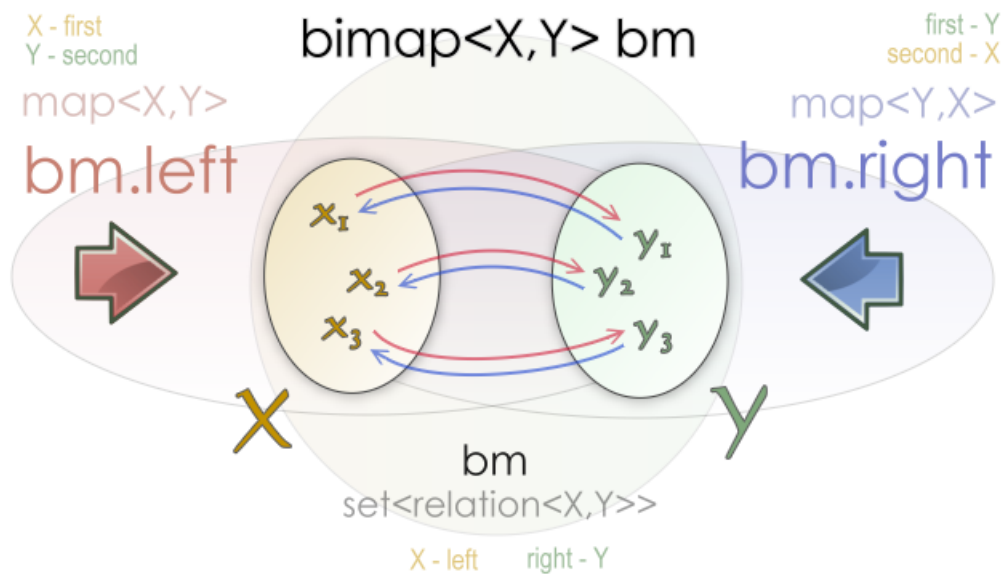
You can see how a bimap container offers three views over the same collection of bidirectional relations.

If we have any generic function that work with maps

---

[1] A type is *signature-compatible* with other type if it has the same signature for functions and metadata. Preconditions, postconditions and the order of operations need not be the same.

```
template< class MapType >
void print_map(const MapType & m)
{
    typedef typename MapType::const_iterator const_iterator;
    for( const_iterator iter = m.begin(), iend = m.end(); iter != iend; ++iter )
    {
        std::cout << iter->first << "-->" << iter->second << std::endl;
    }
}
```

We can use the *left map view* and the *right map view* with it

```
bimap< int, std::string > bm;
...
print_map( bm.left  );
print_map( bm.right );
```

And the output will be

```
1 --> one
2 --> two
...
one --> 1
two --> 2
...
```

Layout of the relation and the pairs of a bimap

The `relation` class represents two related elements. The two values are named left and right to express the symmetry of this type. The bimap pair classes are signature-compatible with `std::pairs`.
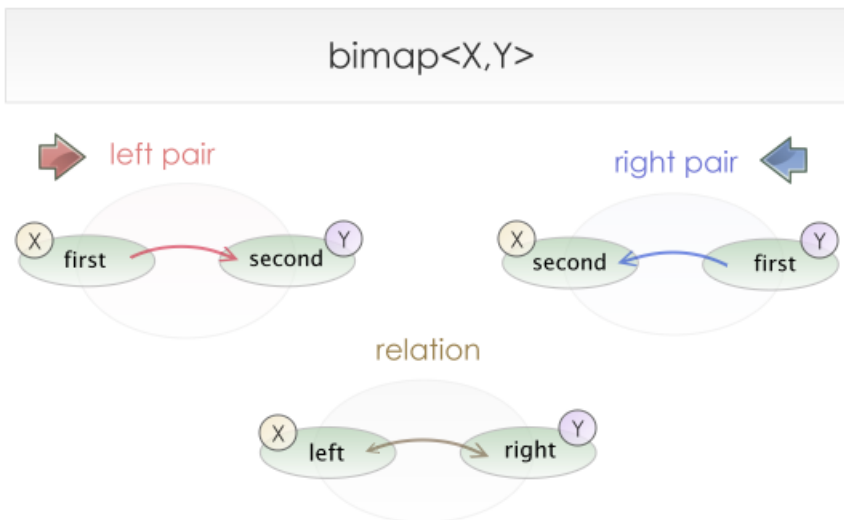
Step by step

A convinience header is avaiable in the boost directory:

```
#include <boost/bimap.hpp>
```

Lets define a bidirectional map between integers and strings:

```
typedef boost::bimap< int, std::string > bm_type;
bm_type bm;
```

The collection of relations view

Remember that `bm` alone can be used as a set of relations. We can insert elements or iterate over them using this view.

```
bm.insert( bm_type::value_type(1, "one" ) );
bm.insert( bm_type::value_type(2, "two" ) );

std::cout << "There are " << bm.size() << "relations" << std::endl;

for( bm_type::const_iterator iter = bm.begin(), iend = bm.end();
    iter != iend; ++iter )
{
    // iter->left  : data : int
    // iter->right : data : std::string

    std::cout << iter->left << " <--> " << iter->right << std::endl;
}
```

The left map view

`bm.left` works like a `std::map< int, std::string >`. We use it in the same way we will use a standard map.

```
❶typedef bm_type::left_map::const_iterator left_const_iterator;

for( left_const_iterator left_iter = bm.left.begin(), iend = bm.left.end();
     left_iter != iend; ++left_iter )
{
    // left_iter->first  : key  : int
    // left_iter->second : data : std::string

    std::cout << left_iter->first << " --> " << left_iter->second << std::endl;
}

❷bm_type::left_const_iterator left_iter = bm.left.find(2);
assert( left_iter->second ==  "two" );

❸bm.left.insert( bm_type::left_value_type( 3, "three" ) );
```

❶   The type of `bm.left` is `bm_type::left_map` and the type of `bm.right` is `bm_type::right_map`
❷   `bm_type::left_-type-` can be used as a shortcut for the more verbose `bm_type::left_map::-type-`
❸   This line produces the same effect of `bm.insert( bm_type::value_type(3,"three") );`

The right map view

`bm.right` works like a `std::map< std::string, int >`. It is important to note that the key is the first type and the data is the second one, exactly as with standard maps.

```
bm_type::right_const_iterator right_iter = bm.right.find("two");

// right_iter->first  : key  : std::string
// right_iter->second : data : int

assert( right_iter->second ==  2 );

assert( bm.right.at("one") == 1 );

bm.right.erase("two");

❶bm.right.insert( bm_type::right_value_type( "four", 4 ) );
```

❶   This line produces the same effect of bm.insert(bm_type::value_type(4,"four"));

Differences with std::map

The main difference between bimap views and their standard containers counterparts is that, because of the bidirectional nature of a bimap, the values stored in it can not be modified directly using iterators. For example, when a std::map<X,Y> iterator is dereferenced the return type is std::pair<const X, Y>, so the following code is valid: m.begin()->second = new_value;. However dereferencing a bimap<X,Y>::left_iterator returns a type that is *signature-compatible* with a std::pair<const X, const Y>

```
bm.left.find(1)->second = "1"; // Compilation error
```

If you insert (1,"one") and (1,"1") in a std::map<int,std::string> the second insertion will have no effect. In a bimap<X,Y> both keys have to remain unique. The insertion may fail in other situtions too. Lets see an example

```
bm.clear();

bm.insert( bm_type::value_type( 1, "one" ) );

bm.insert( bm_type::value_type( 1, "1"   ) ); // No effect!
bm.insert( bm_type::value_type( 2, "one" ) ); // No effect!

assert( bm.size() == 1 );
```

A simple example

Look how you can reuse code that is intend to be used with std::maps, like the print_map function in this example.

Go to source code

```cpp
#include <string>
#include <iostream>

#include <boost/bimap.hpp>

template< class MapType >
void print_map(const MapType & map,
               const std::string & separator,
               std::ostream & os )
{
    typedef typename MapType::const_iterator const_iterator;

    for( const_iterator i = map.begin(), iend = map.end(); i != iend; ++i )
    {
        os << i->first << separator << i->second << std::endl;
    }
}

int main()
{
    // Soccer World cup

    typedef boost::bimap< std::string, int > results_bimap;
    typedef results_bimap::value_type position;

    results_bimap results;
    results.insert( position("Argentina"   ,1) );
    results.insert( position("Spain"       ,2) );
    results.insert( position("Germany"     ,3) );
    results.insert( position("France"      ,4) );

    std::cout << "The number of countries is " << results.size()
              << std::endl;

    std::cout << "The winner is " << results.right.at(1)
              << std::endl
              << std::endl;

    std::cout << "Countries names ordered by their final position:"
              << std::endl;

    // results.right works like a std::map< int, std::string >

    print_map( results.right, ") ", std::cout );

    std::cout << std::endl
              << "Countries names ordered alphabetically along with"
                 "their final position:"
              << std::endl;

    // results.left works like a std::map< std::string, int >

    print_map( results.left, " ends in position ", std::cout );

    return 0;
}
```

The output of this program will be the following:

```
The number of countries is 4

The winner is Argentina

Countries names ordered by their final position:
1) Argentina
2) Spain
3) Germany
4) France

Countries names ordered alphabetically along with their final position:
Argentina ends in position 1
France ends in position 4
Germany ends in position 3
Spain ends in position 2
```

Continuing the journey

For information on function signatures, see any standard library documentation or read the reference section of this documentation.

> **⚠ Caution**
>
> Be aware that a bidirectional map is only signature-compatible with standard containers. Some functions may give different results, such as in the case of inserting a pair into the left map where the second value conflicts with a stored relation in the container. The functions may be slower in a bimap because of the duplicated constraints. It is strongly recommended that you read The full tutorial if you intend to use a bimap in a serious project.

The tutorial

Roadmap

1. Boost.Bimap is intuitive because it is based on the standard template library. New concepts are however presented to extend the standard maps to bidirectional maps. The first step is to gain a firm grasp of the bimap framework. The first section (Discovering the bimap framework) aims to explain this.

2. Boost.Bimap offers much more than just a one-to-one ordered unique bidirectional map. It is possible to control the collection type of each side of the relationship that the bimap represents, giving one-to-many containers, hashed bidirectional containers and others that may be more suitable to the the task at hand. The second section (Controlling collection types) explains how to instantiate a bimap with different collection constraints.

3. The section (The "collection of relations" type) explains how to create new types of bidirectional maps using custom collection types.

4. In the section Differences with standard maps we will learn about the subtle differences between a bimap map view and a standard map.

5. The section Useful functions provides information about functions of a bimap that are not found in the STL.

6. The types of a bimap can be tagged so that each side is accessible by something closer to the problem than left and right. This leads to more readable, self-documenting code. The fourth section (Bimaps with user defined names) shows how to use this feature.

7. The bimap mapping framework allows to disable a view of a bimap, including the standard mapping containers as a particular case. The section Unconstrained Sets explains how they work.

8. The section Additional information explains how to attach information to each relation of a bimap.

9. The final section (Complete Instantiation Scheme) summarizes bimap instantiation and explains how change the allocator type to be used.

Discovering the bimap framework

Interpreting bidirectional maps

One way to interpret bidirectional maps is as a function between two collections of data, lets call them the left and the right collection. An element in this bimap is a relation between an element from the left collection and an element from the right collection. The types of both collections defines the bimap behaviour. We can view the stored data from the left side, as a mapping between keys from the left collection and data from the right one, or from the right side, as a mapping between keys from the right collection and data from the left collection.

Standard mapping framework

Relationships between data in the STL are represented by maps. A standard map is a directed relation of keys from a left collection and data from a right unconstrained collection. The following diagram shows the relationship represented and the user's viewpoint.
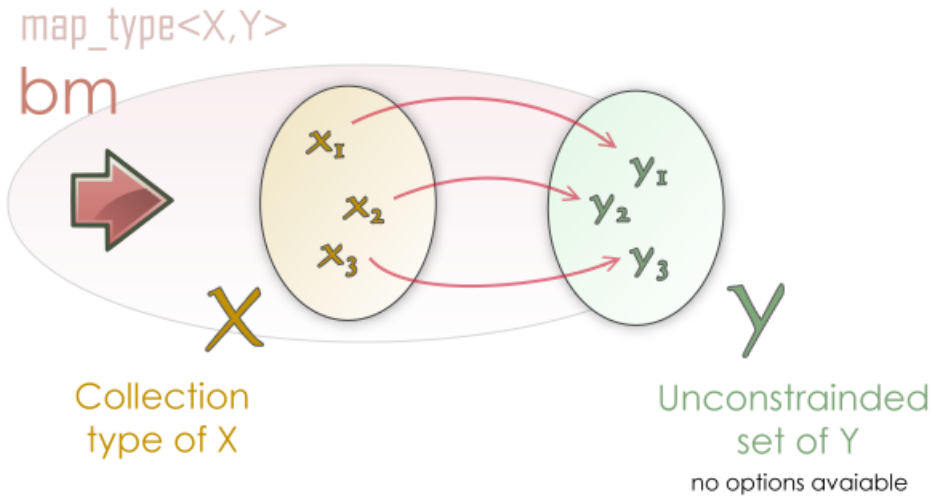
The left collection type depends on the selected map type. For example if the the map type is `std::multimap` the collection type of X is a `multiset_of`. The following table shows the equivalent types for the std associative containers.

Table 1. std associative containers

container	left collection type	right collection type
`map`	`set_of`	no constraints
`multimap`	`multiset_of`	no constraints
`unordered_map`	`unordered_set_of`	no constraints
`unordered_multimap`	`unordered_multiset_of`	no constraints

Bimap mapping framework

Boost.Bimap design is based on the STL, and extends the framework in a natural way. The following diagram represents the new situation.
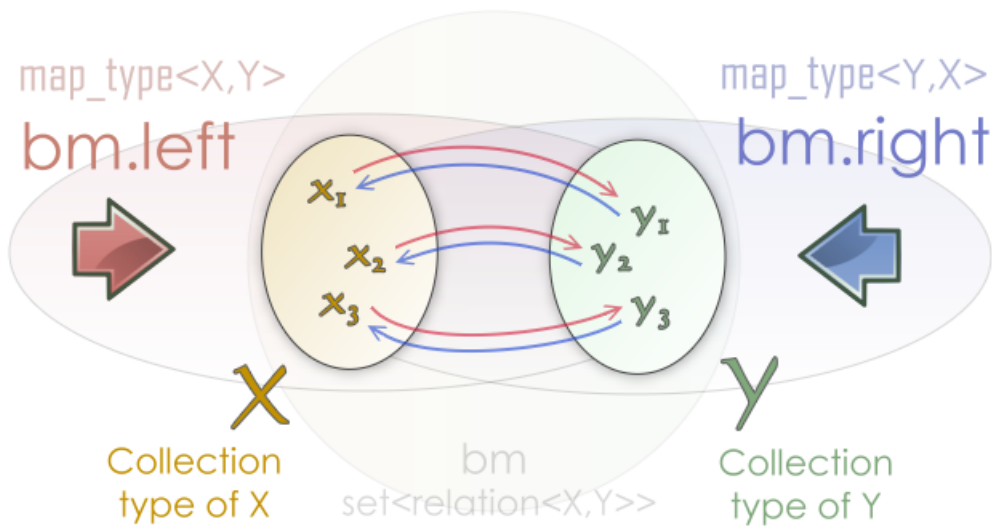
Notice that now the `std::maps` are a particular case of a Boost.Bimap container, where you can view only one side of the relationship and can control the constraints of only one of the collections. Boost.Bimap allows the user to view the relationship from three viewpoints. You can view it from one side, obtaining a `std::map` compatible container, or you can work directly with the whole relation.

The next diagram shows the layout of the relation and pairs of a bimap. It is the one from the *one minute tutorial*
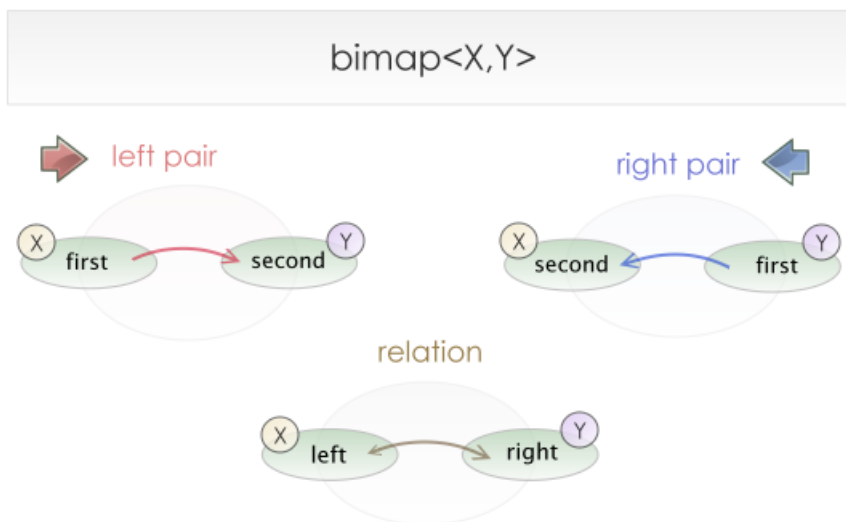
Bimap pairs are signature-compatible with standard pairs but are different from them. As you will see in other sections they can be tagged with user defined names and additional information can be attached to them. You can convert from `std::pairs` to bimap pairs directly but the reverse conversion is not provided. This mean that you can insert elements in a bimap using algorithms like `std::copy` from containers `like std::map`, or use `std::make_pair` to add new elements. However it is best to use `bm.left.insert( bm_type::left_value_type(f,s) )` instead of `bm.insert( std::make_pair(f,s) )` to avoid an extra call to the copy constructor of each type.

The following code snippet shows the relation between a bimap and standard maps.

14

> **Note**
>
> You have to used references to views, and not directly views object. Views cannot be constructed as separate objects from the container they belong to, so the following:
>
> ```
> // Wrong: we forgot the & after bm_type::left_type
> bm_type::left_map lm = bm.left;
> ```
>
> does not compile, since it is trying to construct the view object `lm`. This is a common source of errors in user code.

Go to source code

```cpp
template< class Map, class CompatibleKey, class CompatibleData >
void use_it( Map & m,
             const CompatibleKey  & key,
             const CompatibleData & data )
{
    typedef typename Map::value_type value_type;
    typedef typename Map::const_iterator const_iterator;

    m.insert( value_type(key,data) );
    const_iterator iter = m.find(key);
    if( iter != m.end() )
    {
        assert( iter->first  == key  );
        assert( iter->second == data );

        std::cout << iter->first << " --> " << iter->second;
    }
    m.erase(key);
}

int main()
{
    typedef bimap< set_of<std::string>, set_of<int> > bimap_type;
    bimap_type bm;

    // Standard map
    {
        typedef std::map< std::string, int > map_type;
        map_type m;

        use_it( m, "one", 1 );
    }

    // Left map view
    {
        typedef bimap_type::left_map map_type;
        map_type & m = bm.left;

        use_it( m, "one", 1 );
    }

    // Reverse standard map
    {
        typedef std::map< int, std::string > reverse_map_type;
        reverse_map_type rm;

        use_it( rm, 1, "one" );
    }
```

```
    // Right map view
    {
        typedef bimap_type::right_map reverse_map_type;
        reverse_map_type & rm = bm.right;

        use_it( rm, 1, "one" );
    }

    return 0;
}
```

Controlling collection types

Freedom of choice

As has already been said, in STL maps, you can only control the constraints from one of the collections, namely the one that you are viewing. In Boost.Bimap, you can control both and it is as easy as using the STL.
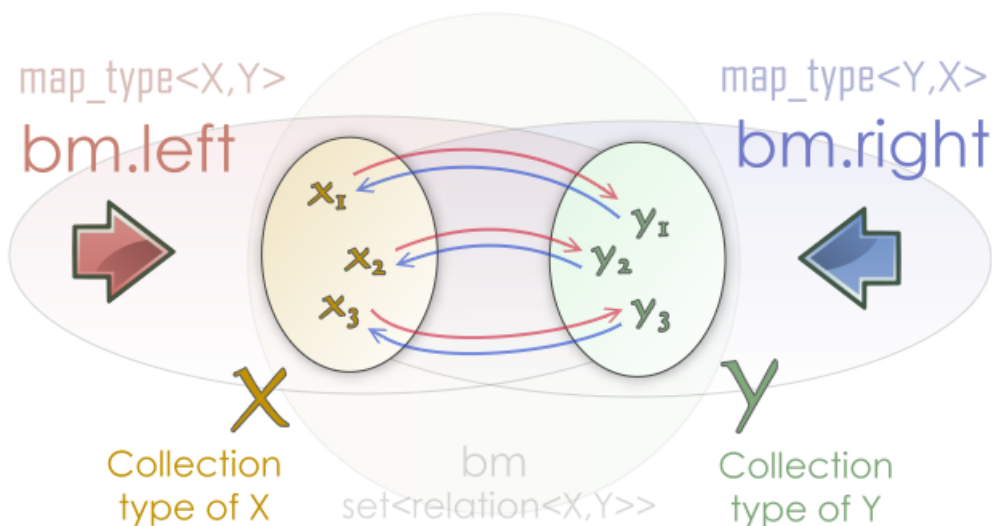
The idea is to use the same constraint names that are used in the standard. If you don't specify the collection type, Boost.Bimap assumes that the collection is a set. The instantiation of a bimap with custom collection types looks like this:

```
typedef bimap< CollectionType_of<A>, CollectionType_of<B> > bm_type;
```

The following is the list of all supported collection types.

Table 2. Collection of Key Types

name	Features	map view type
set_of	*ordered, unique*	map
multiset_of	*ordered*	multimap
unordered_set_of	*hashed, unique*	unordered_map
unordered_multiset_of	*hashed*	unordered_multimap
list_of	*sequenced*	list_map
vector_of	*random access*	vector_map
unconstrained_set_of	*unconstrained*	*can not be viewed*

list_of and vector_of map views are not associated with any existing STL associative containers. They are two examples of unsorted associative containers. unconstrained_set_of allow the user to ignore a view. This will be explained later.
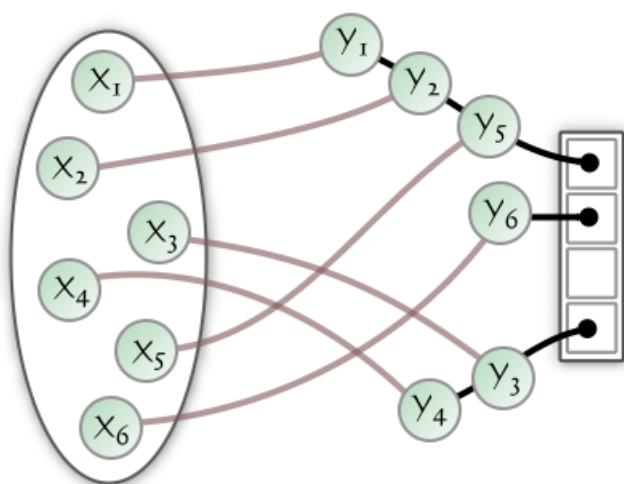
bimap<set_of<X>,unordered_set_of<Y>>
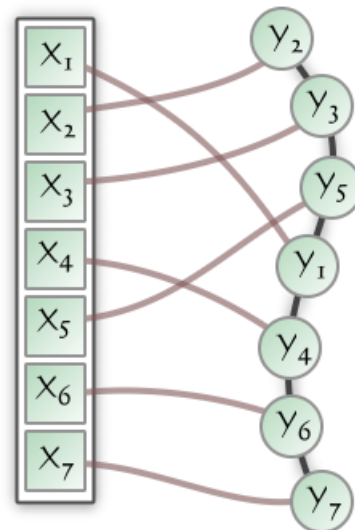
bimap<vector_of<X>,list_of<Y>>

The selection of the collection type affects the possible operations that you can perform with each side of the bimap and the time it takes to do each. If we have:

```
typedef bimap< CollectionType_of<A>, CollectionType_of<B> > bm_type;
bm_type bm;
```

The following now describes the resulting map views of the bidirectional map.

- bm.left is signature-compatible with **LeftMapType**<A,B>

- bm.right is signature-compatible with **RightMapType**<B,A>

Configuration parameters

Each collection type template has different parameters to control its behaviour. For example, in `set_of` specification, you can pass a Functor type that compares two types. All of these parameters are exactly the same as those of the standard library container, except for the allocator type. You will learn later how to change the allocator for a bimap.

The following table lists the meanings of each collection type's parameters.

name	Additional Parameters

`set_of<T,KeyComp>`
 	
`multiset_of<T,KeyComp>`
	KeyComp is a Functor that compares two types using a less-than operator. By default, this is `std::less<T>`.

`unordered_set_of<T,HashFunctor,EqualKey>`
 	
`unordered_multiset_of<T,HashFunctor,EqualKey>`
	HashFunctor converts a `T` object into an `std::size_t` value. By default it is `boost::hash<T>`.
	
	EqualKey is a Functor that tests two types for equality. By default, the equality operator is `std::equal_to<T>`.

`list_of<T>`	No additional parameters.
`vector_of<T>`	No additional parameters.
`unconstrained_set_of<T>`	No additional parameters.

Examples

Countries Populations

We want to store countries populations. The requeriments are:

1. Get a list of countries in decresing order of their populations.

2. Given a countrie, get their population.

Lets create the appropiate bimap.

```
typedef bimap<

    unordered_set_of< std::string >,
    multiset_of< long, std::greater<long> >

> populations_bimap;
```

First of all countries names are unique identifiers, while two countries may have the same population. This is why we choose **multi**`set_of` for populations.

Using a `multiset_of` for population allow us to iterate over the data. Since listing countries ordered by their names is not a requisite, we can use an `unordered_set_of` that allows constant order look up.

And now lets use it in a complete example

Go to source code

```
typedef bimap<

    unordered_set_of< std::string >,
    multiset_of< long, std::greater<long> >

> population_bimap;

typedef population_bimap::value_type population;

population_bimap pop;
pop.insert( population("China",          1321000000) );
pop.insert( population("India",          1129000000) );
pop.insert( population("United States",   301950000) );
pop.insert( population("Indonesia",       234950000) );
pop.insert( population("Brazil",          186500000) );
pop.insert( population("Pakistan",        163630000) );

std::cout << "Countries by their population:" << std::endl;

// First requirement
❶for( population_bimap::right_const_iterator
        i = pop.right.begin(), iend = pop.right.end();
        i != iend ; ++i )
{
    std::cout << i->second << " with " << i->first << std::endl;
}

// Second requirement
❷std::cout << "Population of China: " << pop.left.at("China") << std::endl;
```

❶ The right map view works like a `std::multimap< long, std::string, std::greater<long> >`, We can iterate over it to print the results in the required order.

❷ The left map view works like a `std::unordered_map< std::string, long >`, given the name of the country we can use it to search for the population in constant time

Repetitions counter

We want to count the repetitions for each word in a text and print them in order of appearance.

Go to source code

```
typedef bimap
<
    unordered_set_of< std::string >,
    list_of< counter > ❶

> word_counter;

typedef boost::tokenizer<boost::char_separator<char> > text_tokenizer;

std::string text=
    "Relations between data in the STL are represented with maps."
    "A map is a directed relation, by using it you are representing "
    "a mapping. In this directed relation, the first type is related to "
    "the second type but it is not true that the inverse relationship "
    "holds. This is useful in a lot of situations, but there are some "
    "relationships that are bidirectional by nature.";

// feed the text into the container
word_counter    wc;
text_tokenizer tok(text,boost::char_separator<char>(" \t\n.,;:!?'\"-"));

for( text_tokenizer::const_iterator it = tok.begin(), it_end = tok.end();
     it != it_end ; ++it )
{
    ❷++ wc.left[*it];
}

// list words with counters by order of appearance
❸for( word_counter::right_const_iterator
        wit = wc.right.begin(), wit_end = wc.right.end();

    wit != wit_end; ++wit )
{
    std::cout << wit->second << ": " << wit->first;
}
```

❶   `counter` is an integer that is initialized in zero in the constructor
❷   Because the right collection type is `list_of`, the right data is not used a key and can be modified in the same way as with standard maps.
❸   When we insert the elements using the left map view, the element is inserted at the end of the list.

The collection of relations type

A new point of view

Being able to change the collection type of the bimap relation view is another very important feature. Remember that this view allows the user to see the container as a group of the stored relations. This view has set semantics instead of map semantics.
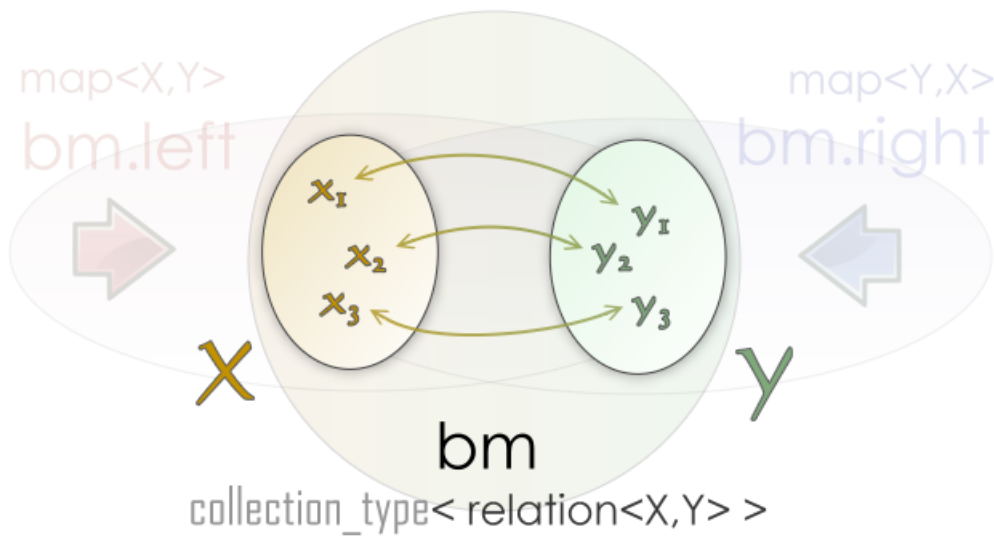
By default, Boost.Bimap will base the collection type of the relation on the type of the left collection. If the left collection type is a set, then the collection type of the relation will be a set with the same order as the left view.

In general, Boost.Bimap users will base the collection type of a relation on the type of the collection on one of the two sides. However there are times where it is useful to give this collection other constraints or simply to order it differently. The user is allowed to choose between:

- left_based

- right_based

- set_of_relation<>

- multiset_of_relation<>

- unordered_set_of_relation<>

- unordered_multiset_of_relation<>

- list_of_relation

- vector_of_relation

- unconstrained_set_of_relation

> **Tip**
>
> The first two options and the last produce faster bimaps, so prefer these where possible.

bimap< X, Y, left_based >   bimap< X, Y, list_of_relation >
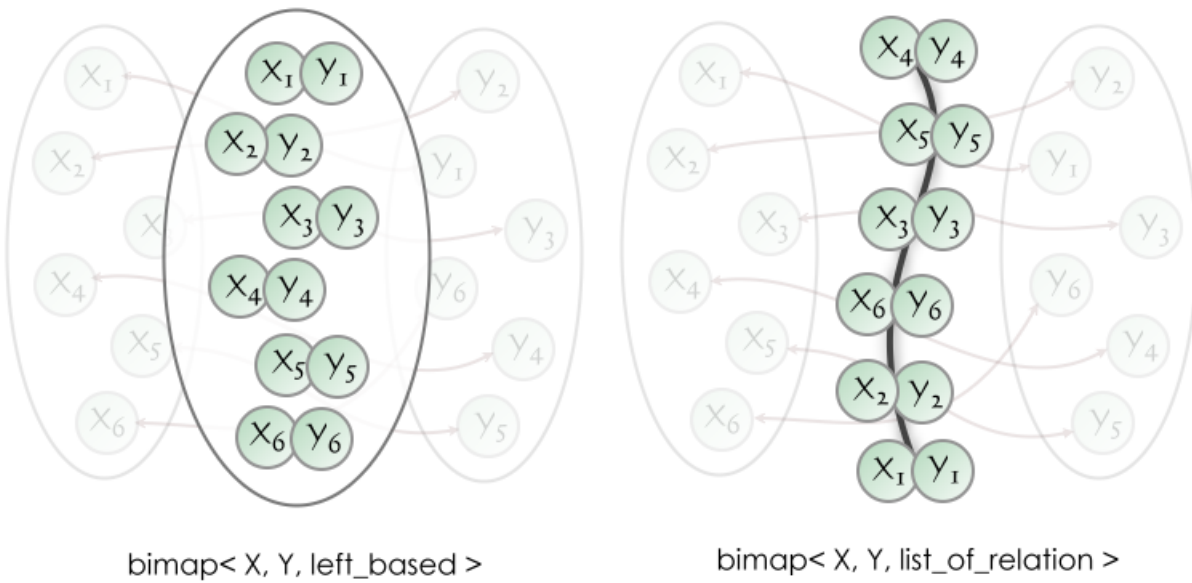
The collection type of relation can be used to create powerful containers. For example, if you need to maximize search speed, then the best bidirectional map possible is one that relates elements from an `unordered_set` to another `unordered_set`. The problem is that this container cannot be iterated. If you need to know the list of relations inside the container, you need another collection type of relation. In this case, a `list_of_relation` is a good choice. The resulting container trades insertion and deletion time against fast search capabilities and the possibility of bidirectional iteration.

Go to source code

```cpp
#include <iostream>
#include <string>
#include <boost/bimap/bimap.hpp>
#include <boost/bimap/list_of.hpp>
#include <boost/bimap/unordered_set_of.hpp>

struct english {};
struct spanish {};

int main()
{
    using namespace boost::bimaps;

    typedef bimap
    <
        unordered_set_of< tagged< std::string, spanish > >,
        unordered_set_of< tagged< std::string, english > >,
        list_of_relation

    > translator;

    translator trans;

    // We have to use `push_back` because the collection of relations is
    // a `list_of_relation`

    trans.push_back( translator::value_type("hola"  ,"hello"   ) );
    trans.push_back( translator::value_type("adios" ,"goodbye" ) );
    trans.push_back( translator::value_type("rosa"  ,"rose"    ) );
    trans.push_back( translator::value_type("mesa"  ,"table"   ) );

    std::cout << "enter a word" << std::endl;
    std::string word;
    std::getline(std::cin,word);

    // Search the queried word on the from index (Spanish)

    translator::map_by<spanish>::const_iterator is
        = trans.by<spanish>().find(word);

    if( is != trans.by<spanish>().end() )
    {
        std::cout << word << " is said "
                  << is->get<english>()
                  << " in English" << std::endl;
    }
    else
    {
        // Word not found in Spanish, try our luck in English

        translator::map_by<english>::const_iterator ie
            = trans.by<english>().find(word);

        if( ie != trans.by<english>().end() )
        {
            std::cout << word << " is said "
                      << ie->get<spanish>()
                      << " in Spanish" << std::endl;
        }
        else
        {
            // Word not found, show the possible translations
```

```
                std::cout << "No such word in the dictionary"       << std::endl;
                std::cout << "These are the possible translations" << std::endl;

                for( translator::const_iterator
                        i = trans.begin(),
                        i_end = trans.end();

                        i != i_end ; ++i )
                {
                    std::cout << i->get<spanish>()
                                << " <---> "
                                << i->get<english>()
                                << std::endl;
                }
            }
        }
        return 0;
}
```

Configuration parameters

Each collection type of relation has different parameters to control its behaviour. For example, in the `set_of_relation` specification, you can pass a Functor type that compares two types. All of the parameters are exactly as in the standard library containers, except for the type, which is set to the bimap relation and the allocator type. To help users in the creation of each functor, the collection type of relation templates takes an mpl lambda expression where the relation type will be evaluated later. A placeholder named `_relation` is available to bimap users.

The following table lists the meaning of the parameters for each collection type of relations.

name	Additional Parameters
`left_based`	Not a template.
`right_based`	Not a template.

`set_of_relation<KeyComp>`
 	
`multiset_of_relation<KeyComp>`
	KeyComp is a Functor that compares two types using less than. By default, the less-than operator is `std::less<_relation>`.

`unordered_set_of_relation<HashFunctor,EqualKey>`
 	
`unordered_multiset_of_relation<HashFunctor,EqualKey>`
	HashFunctor converts the `relation` into an `std::size_t` value. By default it is `boost::hash<_relation>`.
	
	EqualKey is a Functor that tests two relations for equality. By default, the equality operator is `std::equal_to<_relation>`.

`list_of_relation`	Not a template.
`vector_of_relation`	Not a template.
`unconstrained_set_of_relation`	Not a template.

Examples

Consider this example:

```
template< class Rel >
struct RelOrder
{
    bool operator()(Rel ra, Rel rb) const
    {
        return (ra.left+ra.right) < (rb.left+rb.right);
    }
};

typedef bimap
<
        multiset_of< int >,
        multiset_of< int >,
        set_of_relation< RelOrder<_relation> >

> bimap_type;
```

Here the bimap relation view is ordered using the information of both sides. This container will only allow unique relations because `set_of_relation` has been used but the elements in each side of the bimap can be repeated.

```
struct name         {};
struct phone_number {};

typedef bimap
<
    tagged< unordered_multiset_of< string >, name         >,
    tagged< unordered_set_of      < int    >, phone_number >,
    set_of_relation<>

> bimap_type;
```

In this other case the bimap will relate names to phone numbers. Names can be repeated and phone numbers are unique. You can perform quick searches by name or phone number and the container can be viewed ordered using the relation view.

Differences with standard maps

Insertion

Remember that a map can be interpreted as a relation between two collections. In bimaps we have the freedom to change both collection types, imposing constrains in each of them. Some insertions that we give for granted to success in standard maps fails with bimaps. For example:

```
bimap<int,std::string> bm;

bm.left.insert(1,"orange");
bm.left.insert(2,"orange"); // No effect! returns make_pair(iter,false)
```

The insertion will only succeed if it is allowed by all views of the `bimap`. In the next snippet we define the right collection as a multiset, when we try to insert the same two elements the second insertion is allowed by the left map view because both values are different and it is allowed by the right map view because it is a non-unique collection type.

```
bimap<int, multiset_of<std::string> > bm;

bm.left.insert(1,"orange");
bm.left.insert(2,"orange"); // Insertion succeed!
```

If we use a custom collection of relation type, the insertion has to be allowed by it too.

iterator::value_type

The relations stored in the Bimap will not be in most cases modifiable directly by iterators because both sides are used as keys of *key-based* sets. When a `bimap<A,B>` left view iterator is dereferenced the return type is *signature-compatible* with a `std::pair< const A, const B >`. However there are some collection types that are not *key_based*, for example list_of. If a Bimap uses one of these collection types there is no problem with modifying the data of that side. The following code is valid:

```
typedef bimap< int, list_of< std::string > > bm_type;
bm_type bm;
bm.insert( bm_type::relation( 1, "one" ) );
...
bm.left.find(1)->second = "1"; // Valid
```

In this case, when the iterator is dereferenced the return type is *signature-compatible* with a `std::pair<const int, std::string>`.

The following table shows the constness of the dereferenced data of each collection type of:

Side collection type	Dereferenced data
`set_of`	*constant*
`multiset_of`	*constant*
`unordered_set_of`	*constant*
`unordered_multiset_of`	*constant*
`list_of`	*mutable*
`vector_of`	*mutable*
`unconstrained_set_of`	*mutable*

Here are some examples. When dereferenced the iterators returns a type that is *signature-compatible* with these types.

Bimap type	Signature-compatible types

`bimap<A,B>`	`iterator` -> `relation<const A,const B>`
	
	`left_iterator` -> `pair<const A,const B>`
	
	`right_iterator` -> `pair<const B,const A>`

`bimap<multiset_of<A>,unordered_set_of<B> >`	`iterator` -> `relation<const A,const B>`
	
	`left_iterator` -> `pair<const A,const B>`
	
	`right_iterator` -> `pair<const B,const A>`

`bimap<set_of<A>,list_of<B> >`	`iterator` -> `relation<const A,B>`
	
	`left_iterator` -> `pair<const A,B>`
	
	`right_iterator` -> `pair<B,const A>`

`bimap<vector_of<A>,set_of<B> >`	`iterator` -> `relation<A,const B>`
	
	`left_iterator` -> `pair<A,const B>`
	
	`right_iterator` -> `pair<const B,A>`

`bimap<list_of<A>,unconstrained_set_of<B> >`	`iterator` -> `relation<A,B>`
	
	`left_iterator` -> `pair<A,B>`
	
	`right_iterator` -> `pair<B,A>`

operator[] and at()

`set_of` and `unordered_set_of` map views overload `operator[]` to retrieve the associated data of a given key only when the other collection type is a mutable one. In these cases it works in the same way as the standard.

```cpp
bimap< unorderd_set_of< std::string>, list_of<int> > bm;

bm.left["one"] = 1; // Ok
```

The standard defines an access function for `map` and `unordered_map`:

```cpp
const data_type & at(const key_type & k) const;
      data_type & at(const key_type & k);
```

These functions look for a key and returns the associated data value, but throws a `std::out_of_range` exception if the key is not found.

In bimaps the constant version of these functions is given for `set_of` and `unorderd_set_of` map views independently of the other collection type. The mutable version is only provided when the other collection type is mutable.

The following examples shows the behaviour of `at(key)`

Go to source code

```
typedef bimap< set_of< std::string >, list_of< int > > bm_type;
bm_type bm;

try
{
    bm.left.at("one") = 1; // throws std::out_of_range
}
catch( std::out_of_range & e ) {}

assert( bm.empty() );

bm.left["one"] = 1; // Ok

assert( bm.left.at("one") == 1 ); // Ok
```

```
typedef bimap< multiset_of<std::string>, unordered_set_of<int> > bm_type;
bm_type bm;

bm.right[1] = "one"; // compilation error

bm.right.insert( bm_type::right_value_type(1,"one") );

assert( bm.right.at(1) == "one" ); // Ok

try
{
    std::cout << bm.right.at(2); // throws std::out_of_range
}
catch( std::out_of_range & e ) {}

bm.right.at(1) = "1"; // compilation error
```

Complexity of operations

The complexity of some operations is different in bimaps. Read the reference to find the complexity of each function.

Useful functions

Projection of iterators

Iterators can be projected to any of the three views of the bimap. A bimap provides three member functions to cope with projection: `project_left`, `project_right` and `project_up`, with projects iterators to the *left map view*, the *right map view* and the *collection of relations view*. These functions take any iterator from the bimap and retrieve an iterator over the projected view pointing to the same element.

Here is an example that uses projection:

Go to source code

```
typedef bimap<std::string,multiset_of<int,std::greater<int> > > bm_type;

bm_type bm;
bm.insert( bm_type::value_type("John" ,34) );
bm.insert( bm_type::value_type("Peter",24) );
bm.insert( bm_type::value_type("Mary" ,12) );

// Find the name of the next younger person after Peter

bm_type::left_const_iterator name_iter = bm.left.find("Peter");

bm_type::right_const_iterator years_iter = bm.project_right(name_iter);

++years_iter;

std::cout << "The next younger person after Peter is " << years_iter->second;
```

replace and modify

These functions are members of the views of a bimap that are not founded in their standard counterparts.

The `replace` family member functions performs in-place replacement of a given element as the following example shows:

Go to source code

```
typedef bimap< int, std::string > bm_type;
bm_type bm;

bm.insert( bm_type::value_type(1,"one") );

// Replace (1,"one") with (1,"1") using the right map view
{
    bm_type::right_iterator it = bm.right.find("one");

    bool successful_replace = bm.right.replace_key( it, "1" );

    assert( successful_replace );
}

bm.insert( bm_type::value_type(2,"two") );

// Fail to replace (1,"1") with (1,"two") using the left map view
{
    assert( bm.size() == 2 );

    bm_type::left_iterator it = bm.left.find(1);

    bool successful_replace = bm.left.replace_data( it, "two" );

    ❶assert( ! successful_replace );
    assert( bm.size() == 2 );
}
```

❶   `it` is still valid here, and the bimap was left unchanged

`replace` functions performs this substitution in such a manner that:

• The complexity is constant time if the changed element retains its original order with respect to all views; it is logarithmic otherwise.

• Iterator and reference validity are preserved.

- The operation is strongly exception-safe, i.e. the `bimap` remains unchanged if some exception (originated by the system or the user's data types) is thrown.

`replace` functions are powerful operations not provided by standard STL containers, and one that is specially handy when strong exception-safety is required.

The observant reader might have noticed that the convenience of replace comes at a cost: namely the whole element has to be copied *twice* to do the updating (when retrieving it and inside `replace`). If elements are expensive to copy, this may be quite a computational cost for the modification of just a tiny part of the object. To cope with this situation, Boost.Bimap provides an alternative updating mechanism: `modify` functions.

`modify` functions accepts a functor (or pointer to function) taking a reference to the data to be changed, thus eliminating the need for spurious copies. Like `replace` functions, `modify` functions does preserve the internal orderings of all the indices of the `bimap`. However, the semantics of modify functions are not entirely equivalent to replace functions. Consider what happens if a collision occurs as a result of modifying the element, i.e. the modified element clashes with another with respect to some unique view. In the case of `replace` functions, the original value is kept and the method returns without altering the container, but `modify` functions cannot afford such an approach, since the modifying functor leaves no trace of the previous value of the element. Integrity constraints thus lead to the following policy: when a collision happens in the process of calling a modify functions, the element is erased and the method returns false. This difference in behavior between `replace` and `modify` functions has to be considered by the programmer on a case-by-case basis.

Boost.Bimap defines new placeholders named `_key` and `_data` to allow a sounder solution. You have to include `<boost/bimap/support/lambda.hpp>` to use them.

[Go to source code](#)

```cpp
typedef bimap< int, std::string > bm_type;
bm_type bm;
bm.insert( bm_type::value_type(1,"one") );

// Modify (1,"one") to (1,"1") using the right map view
{
    bm_type::right_iterator it = bm.right.find("one");

    bool successful_modify = bm.right.modify_key( it , _key = "1" );

    assert( successful_modify );
}

bm.insert( bm_type::value_type(2,"two") );

// Fail to modify (1,"1") to (1,"two") using the left map view
{
    assert( bm.size() == 2 );

    bm_type::left_iterator it = bm.left.find(1);

    bool successful_modify = bm.left.modify_data( it, _data = "two" );

    ❶assert( ! successful_modify );
    assert( bm.size() == 1 );
}
```

❶ `it` is not longer valid and `(1,"1")` is removed from the bimap

Retrieval of ranges

Standard `lower_bound` and `upper_bound` functions can be used to lookup for all the elements in a given range.

Suppose we want to retrieve the elements from a `bimap<int,std::string>` where the left value is in the range `[20,50]`

```
typedef bimap<int,std::string> bm_type;
bm_type bm;

// ...

bm_type::left_iterator iter_first  = bm.left.lower_bound(20);
bm_type::left_iterator iter_second = bm.left.upper_bound(50);

// range [iter_first,iter_second) contains the elements in [20,50]
```

Subtle changes to the code are required when strict inequalities are considered. To retrieve the elements greater than 20 and less than 50, the code has to be rewritten as

```
bm_type::left_iterator iter_first  = bm.left.upper_bound(20);
bm_type::left_iterator iter_second = bm.left.lower_bound(50);

// range [iter_first,iter_second) contains the elements in (20,50)
```

To add to this complexity, the careful programmer has to take into account that the lower and upper bounds of the interval searched be compatible: for instance, if the lower bound is 50 and the upper bound is 20, the iterators `iter_first` and `iter_second` produced by the code above will be in reverse order, with possibly catastrophic results if a traversal from `iter_first` to `iter_second` is tried. All these details make range searching a tedious and error prone task.

The range member function, often in combination with lambda expressions, can greatly help alleviate this situation:

```
typedef bimap<int,std::string> bm_type;
bm_type bm;

// ...

❶bm_type::left_range_type r;

❷r = bm.left.range( 20 <= _key, _key <= 50 ); // [20,50]

r = bm.left.range( 20 <  _key, _key <  50 ); // (20,50)

r = bm.left.range( 20 <= _key, _key <  50 ); // [20,50)
```

❶   `range_type` is a handy typedef equal to `std::pair<iterator,iterator>`. `const_range_type` is provided too, and it is equal to `std::pair<const_iterator,const_iterator>`

❷   `_key` is a Boost.Lambda placeholder. To use it you have to include `<boost/bimap/support/lambda.hpp>`

`range` simply accepts predicates specifying the lower and upper bounds of the interval searched. Please consult the reference for a detailed explanation of the permissible predicates passed to range.

One or both bounds can be omitted with the special unbounded marker:

```
r = bm.left.range( 20 <= _key, unbounded ); // [20,inf)

r = bm.left.range( unbounded , _key < 50 ); // (-inf,50)

❶r = bm.left.range( unbounded , unbounded ); // (-inf,inf)
```

❶   This is equivalent to std::make_pair(s.begin(),s.end())

Go to source code

Bimaps with user defined names

In the following example, the library user inserted comments to guide future programmers:

Go to source code

```cpp
typedef bimap
<
    multiset_of<std::string>,
    int

> People;

People people;

// ...

int user_id;
std::cin >> user_id;

// people.right : map<id,name>

People::right_const_iterator id_iter = people.right.find(user_id);
if( id_iter != people.right.end() )
{
    // first  : id
    // second : name

    std::cout << "name: " << id_iter->second << std::endl
              << "id:   " << id_iter->first  << std::endl;
}
else
{
    std::cout << "Unknown id, users are:" << std::endl;

    // people.left : map<name,id>

    for( People::left_const_iterator
            name_iter = people.left.begin(),
            iend      = people.left.end();

         name_iter != iend; ++name_iter )
    {
        // first  : name
        // second : id

        std::cout << "name: " << name_iter->first  << std::endl
                  << "id:   " << name_iter->second << std::endl;
    }
}
```

In Boost.Bimap there is a better way to document the code and in the meantime helping you to write more mantainable and readable code. You can tag the two collections of the bimap so they can be accessed by more descriptive names.

A tagged type is a type that has been labelled using a tag. A tag is any valid C++ type. In a bimap, the types are always tagged. If you do not specify your own tag, the container uses `member_at::left` and `member_at::right` to tag the left and right sides respectively. In order to specify a custom tag, the type of each side has to be tagged. Tagging a type is very simple:

```
typedef tagged< int, a_tag > tagged_int;
```

Now we can rewrite the example:

Go to source code

```cpp
struct id   {}; // Tag for the identification number
struct name {}; // Tag for the name of the person

typedef bimap
<
                tagged< int        , id   > ,
    multiset_of< tagged< std::string, name > >

> People;

People people;

// ...

int user_id;
std::cin >> user_id;

People::map_by<id>::const_iterator id_iter = people.by<id>().find(user_id);
if( id_iter != people.by<id>().end() )
{
    std::cout << "name: " << id_iter->get<name>() << std::endl
              << "id:   " << id_iter->get<id>()   << std::endl;
}
else
{
    std::cout << "Unknown id, users are:" << std::endl;

    for( People::map_by<name>::const_iterator
            name_iter = people.by<name>().begin(),
            iend      = people.by<name>().end();

        name_iter != iend; ++name_iter )
    {
        std::cout << "name: " << name_iter->get<name>() << std::endl
                  << "id:   " << name_iter->get<id>()   << std::endl;
    }
}
```

Here is a list of common structures in both tagged and untagged versions. Remember that when the bimap has user defined tags you can still use the untagged version structures.

```cpp
struct Left  {};
struct Right {};
typedef bimap<
        multiset_of< tagged< int, Left  > >,
    unordered_set_of< tagged< int, Right > >
> bm_type;

bm_type bm;

//...

bm_type::iterator       iter       = bm.begin();
bm_type::left_iterator  left_iter  = bm.left.begin();
bm_type::right_iterator right_iter = bm.right.begin();
```

Table 3. Equivalence of expresions using user defined names

Untagged version	Tagged version
`bm.left`	`bm.by<Left>()`
`bm.right`	`bm.by<Right>()`
`bm_type::left_map`	`bm::map_by<Left>::type`
`bm_type::right_value_type`	`bm::map_by<Right>::value_type`
`bm_type::left_iterator`	`bm::map_by<Left>::iterator`
`bm_type::right_const_iterator`	`bm::map_by<Right>::const_iterator`
`iter->left`	`iter->get<Left>()`
`iter->right`	`iter->get<Right>()`
`left_iter->first`	`left_iter->get<Left>()`
`left_iter->second`	`left_iter->get<Right>()`
`right_iter->first`	`right_iter->get<Right>()`
`right_iter->second`	`right_iter->get<Left>()`
`bm.project_left(iter)`	`bm.project<Left>(iter)`
`bm.project_right(iter)`	`bm.project<Right>(iter)`

Unconstrained Sets

Unconstrained sets allow the user to disable one of the views of a bimap. Doing so makes the bimap operations execute faster and reduces memory consumption. This completes the bidirectional mapping framework by including unidirectional mappings as a particular case.

Unconstrained sets are useful for the following reasons:

- A bimap type has stronger guarantees than its standard equivalent, and includes some useful functions (replace, modify) that the standard does not have.

- You can view the mapping as a collection of relations.

- Using this kind of map makes the code very extensible. If, at any moment of the development, the need to perform searches from the right side of the mapping arises, the only necessary change is to the `typedef`.

Given this bimap instance,

```
typedef bimap< std::string, unconstrained_set_of<int> > bm_type;
typedef bm_type::left_map map_type;

bm_type bm;
map_type & m = bm.left;
```

or this standard map one

```
typedef std::map< std::string, int > map_type;

map_type m;
```

The following code snippet is valid

```
m["one"] = 1;

assert( m.find("one") != m.end() );

for( map_type::iterator i = m.begin(), iend = m.end(); i != iend; ++i )
{
    ❶++(i->second);
}

m.erase("one");
```

❶ The right collection of the bimap is mutable so its elements can be modified using iterators.

But using a bimap has some benefits

```
typedef map_type::const_iterator const_iterator;
typedef std::pair<const_iterator,const_iterator> const_range;

❶const_range r = m.range( "one" <= _key, _key <= "two" );
for( const_iterator i = r.first; i != r.second; ++i )
{
    std::cout << i->first << "-->" << i->second << std::endl;
}

m.modify_key( m.begin(), _key = "1" );
```

❶ This range is a model of BidirectionalRange, read the docs of Boost.Range for more information.

Go to source code

Additional information

Bidirectional maps may have associated information about each relation. Suppose we want to represent a books and author bidirectional map.

35

```
typedef bimap<

    multiset_of< std::string >, // author
        set_of< std::string >  // title

> bm_type;
typedef bm_type::value_type book;

bm_type bm;

bm.insert( book( "Bjarne Stroustrup"   , "The C++ Programming Language" ) );
bm.insert( book( "Scott Meyers"        , "Effective C++"                ) );
bm.insert( book( "Andrei Alexandrescu" , "Modern C++ Design"            ) );

// Print the author of Modern C++
std::cout << bm.right.at( "Modern C++ Design" );
```

Suppose now that we want to store abstract of each book. We have two options:

1. Books name are unique identifiers, so we can create a separate `std::map< string, string >` that relates books names with abstracts.

2. We can use **Boost.MultiIndex** for the new beast.

Option 1 is the wrong approach, if we go this path we lost what bimap has won us. We now have to maintain the logic of two interdependent containers, there is an extra string stored for each book name, and the performance will be worse. This is far away from being a good solution.

Option 2 is correct. We start thinking books as entries in a table. So it makes sense to start using Boost.MultiIndex. We can then add the year of publication, the price, etc... and we can index this new items too. So Boost.MultiIndex is a sound solution for our problem.

The thing is that there are cases where we want to maintain bimap semantics (use `at()` to find an author given a book name and the other way around) and add information about the relations that we are sure we will not want to index later (like the abstracts). Option 1 is not possible, option 2 neither.

Boost.Bimap provides support for this kind of situations by means of an embedded information member. You can pass an extra parameter to a bimap: `with_info< InfoType >` and an `info` member of type `InfoType` will appear in the relation and bimap pairs.
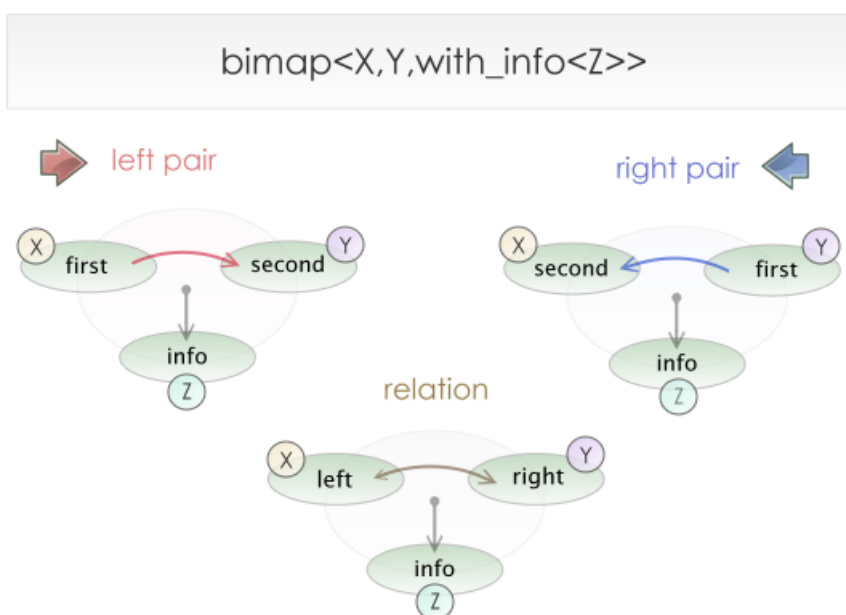
Relations and bimap pairs constructors will take an extra argument. If only two arguments are used, the information will be initialized with their default constructor.

```cpp
typedef bimap<

    multiset_of< std::string >, // author
         set_of< std::string >, // title

      with_info< std::string >  // abstract

> bm_type;
typedef bm_type::value_type book;

bm_type bm;

bm.insert(

    book( "Bjarne Stroustrup"    , "The C++ Programming Language",

         "For C++ old-timers, the first edition of this book is"
         "the one that started it all—the font of our knowledge." )
);


// Print the author of the bible
std::cout << bm.right.at("The C++ Programming Language");

// Print the abstract of this book
bm_type::left_iterator i = bm.left.find("Bjarne Stroustrup");
std::cout << i->info;
```

Contrary to the two key types, the information will be mutable using iterators.

```cpp
i->info += "More details about this book";
```

A new function is included in *unique* map views: `info_at(key)`, that mimics the standard `at(key)` function but returned the associated information instead of the data.

```cpp
// Print the new abstract
std::cout << bm.right.info_at("The C++ Programming Language");
```

The info member can be tagged just as the left or the right member. The following is a rewrite of the above example using user defined names:

```
typedef bimap<

    multiset_of< tagged< std::string, author   > >,
        set_of< tagged< std::string, title    > >,

      with_info< tagged< std::string, abstract > >

> bm_type;
typedef bm_type::value_type book;

bm_type bm;

bm.insert(

    book( "Bjarne Stroustrup"   , "The C++ Programming Language",

          "For C++ old-timers, the first edition of this book is"
          "the one that started it all—the font of our knowledge." )
);

// Print the author of the bible
std::cout << bm.by<title>().at("The C++ Programming Language");

// Print the abstract of this book
bm_type::map_by<author>::iterator i = bm.by<author>().find("Bjarne Stroustrup");
std::cout << i->get<abstract>();

// Contrary to the two key types, the information will be mutable
// using iterators.

i->get<abstract>() += "More details about this book";

// Print the new abstract
std::cout << bm.by<title>().info_at("The C++ Programming Language");
```

Go to source code

Complete instantiation scheme

To summarize, this is the complete instantiation scheme.

```
typedef bimap
<
    LeftCollectionType, RightCollectionType

    [ , SetTypeOfRelation  ]  // Default to left_based
    [ , with_info< Info >  ]  // Default to no info
    [ , Allocator          ]  // Default to std::allocator<>

> bm;
```

`{Side}CollectionType` can directly be a type. This defaults to `set_of<Type>`, or can be a `{CollectionType}_of<Type>` specification. Additionally, the type of this two parameters can be tagged to specify user defined names instead of the usual `member_at::-Side-` tags.

The possibles way to use the first parameter are:

```
bimap< Type, R >
```

• Left type: `Type`

- Left collection type: `set_of< Type >`

- Left tag: `member_at::left`

```
bimap< {CollectionType}_of< Type >, R >
```

- Left type: `Type`

- Left collection type: `{CollectionType}_of< LeftType >`

- Left tag: `member_at::left`

```
bimap< tagged< Type, Tag >, R >
```

- Left type: `Type`

- Left collection type: `set_of< LeftType >`

- Left tag: `Tag`

```
bimap< {CollectionType}_of< tagged< Type, Tag > >, R >
```

- Left type: `Type`

- Left collection type: `{CollectionType}_of< LeftType >`

- Left tag: `Tag`

The same options are available for the second parameter.

The last three parameters are used to specify the collection type of the relation, the information member and the allocator type.

If you want to specify a custom allocator type while relying on the default value of CollectionTypeOfRelation, you can do so by simply writing `bimap<LeftKeyType, RightKeyType, Allocator>`. Boost.Bimap's internal machinery detects that the third parameter in this case does not refer to the relation type but rather to an allocator.

The following are the possible ways of instantiating the last three parameters of a bimap. You can ignore some of the parameter but the order must be respected.

```
bimap< L, R >
```

- set_of_relation_type: based on the left key type

- info: no info

- allocator: std::allocator

```
bimap< L, R ,SetOfRelationType>
```

- set_of_relation_type: SetOfRelationType

- info: no info

- allocator: std::allocator

```
bimap< L, R , SetOfRelationType, with_info<Info> >
```

39

- set_of_relation_type: SetOfRelationType

- info: Info

- allocator: std::allocator

```
bimap< L, R , SetOfRelationType, with_info<Info>, Allocator>
```

- set_of_relation_type: SetOfRelationType

- info: Info

- allocator: Allocator

```
bimap< L, R , SetOfRelationType, Allocator>
```

- set_of_relation_type: SetOfRelationType

- info: no info

- allocator: Allocator

```
bimap< L, R , with_info<Info> >
```

- set_of_relation_type: based on the left key type

- info: Info

- allocator: std::allocator

```
bimap< L, R , with_info<Info>, Allocator>
```

- set_of_relation_type: based on the left key type

- allocator: Allocator

```
bimap< L, R , Allocator>
```

- set_of_relation_type: based on the left key type

- info: no info

- allocator: Allocator

Bimap and Boost

Bimap and MultiIndex

MISC - **M**ulti-**I**ndex **S**pecialized **C**ontainers

> *Let's be generic, construct frameworks, describe the world in an unified way...*
>
> *No!, it is better to be specialized, design easy-to-use components, offer plug-and-play objects...*
>
> **Why not take advantage of the best of both worlds?**
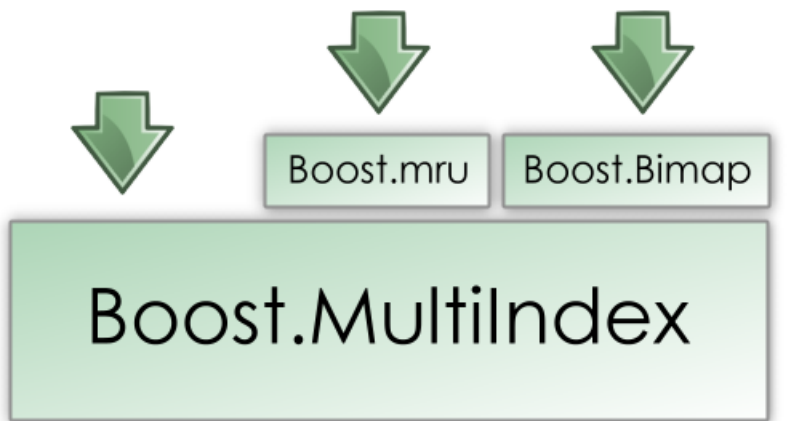
With Boost.Bimap, you can build associative containers in which both types can be used as key. There is a library in Boost that already allows the creation of this kind of container: Boost.MultiIndex. It offers great flexibility and lets you construct almost any container that you could dream of. The framework is very clean. You migh want to read this library's tutorial to learn about the power that has been achieved.

But generality comes at a price: the interface that results might not be the best for every specialization. People may end up wrapping a B.MI container in its own class every time they want to use it as a bidirectional map. Boost.Bimap takes advantage of the narrower scope to produce a better interface for bidirectional maps [2]. There is no learning curve if you know how to use standard containers. Great effort was put into mapping the naming scheme of the STL to Boost.Bimap. The library is designed to match the common STL containers.

Boost.MultiIndex is, in fact, the core of the bimap container.

However, Boost.Bimap do not aim to tackle every problem with two indexed types. There exist some problems that are better modelled with Boost.MultiIndex.

Problem I - An employee register

Store an ID and a name for an employee, with fast search on each member.

This type of problem is better modelled as a database table, and **Boost.MultiIndex** is the preferred choice. It is possible that other data will need to be indexed later.

---

[2] In the same fashion, Boost.MRU will allow the creation of *most recent updated* aware containers, hiding the complexity of Boost.MultiIndex.

> **Problem II - A partners container**
>
> *Store the names of couples and be able to get the name of a person's partner.*
>
> This problem is better modelled as a collection of relations, and **Boost.Bimap** fits nicely here.

You can also read Additional Information for more information about the relation of this two libraries.

Boost Libraries that work well with Boost.Bimap

Introduction

Name	Description	author	Purpose
Boost.Serialization	Serialization for persistence and marshalling	Robert Ramey	Serialization support for bimap containers and iterators
Boost.Assign	Filling containers with constant or generated data has never been easier	Thorsten Ottosen	Help to fill a bimap or views of it
Boost.Hash	A TR1 hash function object that can be extended to hash user defined types	Daniel James	Default hashing function
Boost.Lambda	Define small unnamed function objects at the actual call site, and more	from Jaakko Järvi, Gary Powell	Functors for modify, range, lower_bound and upper_bound
Boost.Range	A new infrastructure for generic algorithms that builds on top of the new iterator concepts	Thorsten Ottosen	Range based algorithms
Boost.Foreach	BOOST_FOREACH macro for easily iterating over the elements of a sequence	Eric Niebler	Iteration
Boost.Typeof	Typeof operator emulation	Arkadiy Vertleyb, Peder Holt	Using BOOST_AUTO while we wait for C++0x
Boost.Xpressive	Regular expressions that can be written as strings or as expression templates	Eric Niebler	Help to fill a bimap from a string
Boost.PropertyMap	Concepts defining interfaces which map key objects to value objects	Jeremy Siek	Integration with BGL

Boost.Serialization

A bimap can be archived and retrieved by means of the Boost.Serialization Library. Both regular and XML archives are supported. The usage is straightforward and does not differ from that of any other serializable type. For instance:

Go to source code

```
typedef bimap< std::string, int > bm_type;

// Create a bimap and serialize it to a file
{
    bm_type bm;
    bm.insert( bm_type::value_type("one",1) );
    bm.insert( bm_type::value_type("two",2) );

    std::ofstream ofs("data");
    boost::archive::text_oarchive oa(ofs);

    oa << const_cast<const bm_type&>(bm); ❶

    ❷const bm_type::left_iterator left_iter = bm.left.find("two");
    oa << left_iter;

    const bm_type::right_iterator right_iter = bm.right.find(1);
    oa << right_iter;
}

// Load the bimap back
{
    bm_type bm;

    std::ifstream ifs("data", std::ios::binary);
    boost::archive::text_iarchive ia(ifs);

    ia >> bm;

    assert( bm.size() == 2 );

    bm_type::left_iterator left_iter;
    ia >> left_iter;

    assert( left_iter->first == "two" );

    bm_type::right_iterator right_iter;
    ia >> right_iter;

    assert( right_iter->first == 1 );
}
```

❶   We must do a const cast because Boost.Serialization archives only save const objects. Read Boost.Serializartion docs for the rationale behind this decision

❷   We can only serialize iterators if the bimap was serialized first. Note that the const cast is not requiered here because we create our iterators as const.

Serialization capabilities are automatically provided by just linking with the appropriate Boost.Serialization library module: it is not necessary to explicitly include any header from Boost.Serialization, apart from those declaring the type of archive used in the process. If not used, however, serialization support can be disabled by globally defining the macro BOOST_BIMAP_DISABLE_SERIALIZATION. Disabling serialization for Boost.MultiIndex can yield a small improvement in build times, and may be necessary in those defective compilers that fail to correctly process Boost.Serialization headers.

Warning

Boost.Bimap and Boost.MultiIndex share a lot of serialization code. The macro BOOST_BIMAP_DISABLE_SERIALIZATION disables serialization in **both** libraries. The same happens when BOOST_MULTI_INDEX_DISABLE_SERIALIZATION is defined.

Retrieving an archived bimap restores not only the elements, but also the order they were arranged in the views of the container. There is an exception to this rule, though: for unordered sets, no guarantee is made about the order in which elements will be iterated in the restored container; in general, it is unwise to rely on the ordering of elements of a hashed view, since it can change in arbitrary ways during insertion or rehashing --this is precisely the reason why hashed indices and TR1 unordered associative containers do not define an equality operator.

Iterators of a bimap can also be serialized. Serialization of an iterator must be done only after serializing its corresponding container.

Boost.Assign

The purpose of this library is to make it easy to fill containers with data by overloading operator,() and operator()(). These two operators make it possible to construct lists of values that are then copied into a container.

These lists are particularly useful in learning, testing, and prototyping situations, but can also be handy otherwise. The library comes with predefined operators for the containers of the standard library, but most functionality will work with any standard compliant container. The library also makes it possible to extend user defined types so for example a member function can be called for a list of values instead of its normal arguments.

Boost.Assign can be used with bimap containers. The views of a bimap are signature-compatible with their standard counterparts, so we can use other Boost.Assign utilities with them.

[Go to source code](#)

```cpp
typedef bimap< multiset_of< int >, list_of< std::string > > bm_type;

// We can use assign::list_of to initialize the container.

bm_type bm = assign::list_of< bm_type::relation > ❶
    ( 1, "one"   )
    ( 2, "two"   )
    ( 3, "three" );

// The left map view is a multiset, again we use insert

assign::insert( bm.left )
    ( 4, "four" )
    ( 5, "five" )
    ( 6, "six"  );

// The right map view is a list so we use push_back here
// Note the order of the elements in the list!

assign::push_back( bm.right )
    ( "seven" , 7 )
    ( "eight" , 8 );

assign::push_front( bm.right )
    ( "nine"  ,  9 )
    ( "ten"   , 10 )
    ( "eleven", 11 );

// Since it is left_based the main view is a multiset, so we use insert

assign::insert( bm )
    ( 12, "twelve"   )
    ( 13, "thirteen" );
```

❶ Note that `bm_type::relation` has to be used instead of `bm_type::value_type`. Contrary to `value_type`, `relation` type stores the elements as non const, a requirement of `assign::list_of`

Boost.Hash

The hash function is the very core of the fast lookup capabilities of the unordered sets: a hasher is just a Unary Function returning an std::size_t value for any given key. In general, it is impossible that every key map to a different hash value, for the space of keys can be greater than the number of permissible hash codes: what makes for a good hasher is that the probability of a collision (two different keys with the same hash value) is as close to zero as possible.

This is a statistical property depending on the typical distribution of keys in a given application, so it is not feasible to have a general-purpose hash function with excellent results in every possible scenario; the default value for this parameter uses Boost.Hash, which often provides good enough results.

Boost.Hash can be extended for custom data types, enabling to use the default parameter of the unordered set types with any user types.

Boost.Lambda

The Boost Lambda Library (BLL in the sequel) is a C++ template library, which implements form of lambda abstractions for C++. The term originates from functional programming and lambda calculus, where a lambda abstraction defines an unnamed function. Lambda expressions are very useful to construct the function objects required by some of the functions in a bimap view.

Boost.Bimap defines new placeholders in `<boost/bimap/support/lambda.hpp>` to allow a sounder solution. The placeholders are named _key and _data and both are equivalent to boost::lambda::_1. There are two reasons to include this placeholders: the code looks better with them and they avoid the clash problem between lambda::_1 and boost::_1 from Boost.Bind.

Go to source code

```
typedef bimap< std::string, int > bm_type;

bm_type bm;
bm.insert( bm_type::value_type("one",1) );
bm.insert( bm_type::value_type("two",2) );

bm.right.range( 5 < _key, _key < 10 );

bm.left.modify_key( bm.left.find("one"), _key = "1" );

bm.left.modify_data( bm.left.begin(), _data *= 10 );
```

Boost.Range

Boost.Range is a collection of concepts and utilities that are particularly useful for specifying and implementing generic algorithms. Generic algorithms have so far been specified in terms of two or more iterators. Two iterators would together form a range of values that the algorithm could work on. This leads to a very general interface, but also to a somewhat clumsy use of the algorithms with redundant specification of container names. Therefore we would like to raise the abstraction level for algorithms so they specify their interface in terms of Ranges as much as possible.

As Boost.Bimap views are signature-compatible with their standard container counterparts, they are compatible with the concept of a range. As an additional feature, ordered bimap views offer a function named `range` that allows a range of values to be obtained.

If we have some generic functions that accepts ranges:

```
template< class ForwardReadableRange, class UnaryFunctor >
UnaryFunctor for_each(const ForwardReadableRange & r, UnaryFunctor func)
{
    typedef typename
    boost::range_const_iterator<ForwardReadableRange>::type const_iterator;

    for(const_iterator i= boost::begin(r), iend= boost::end(r); i!=iend; ++i )
    {
        func(*i);
    }

    return func;
}

template< class ForwardReadableRange, class Predicate >
typename boost::range_difference<ForwardReadableRange>::type
    count_if(const ForwardReadableRange & r, Predicate pred)
{
    typedef typename
    boost::range_const_iterator<ForwardReadableRange>::type const_iterator;

    typename boost::range_difference<ForwardReadableRange>::type c = 0;

    for( const_iterator i = boost::begin(r), iend = boost::end(r); i != iend; ++i )
    {
        if( pred(*i) ) ++c;
    }

    return c;
}
```

We can use them with Boost.Bimap with the help of the `range` function.

```cpp
struct pair_printer
{
    pair_printer(std::ostream & o) : os(o) {}
    template< class Pair >
    void operator()(const Pair & p)
    {
        os << "(" << p.first << "," << p.second << ")";
    }
    private:
    std::ostream & os;
};

struct second_extractor
{
    template< class Pair >
    const typename Pair::second_type & operator()(const Pair & p)
    {
        return p.second;
    }
};

int main()
{
    typedef bimap< double, multiset_of<int> > bm_type;

    bm_type bm;
    bm.insert( bm_type::value_type(2.5 , 1) );
    bm.insert( bm_type::value_type(3.1 , 2) );
    //...
    bm.insert( bm_type::value_type(6.4 , 4) );
    bm.insert( bm_type::value_type(1.7 , 2) );

    // Print all the elements of the left map view

    for_each( bm.left, pair_printer(std::cout) );

    // Print a range of elements of the right map view

    for_each( bm.right.range( 2 <= _key, _key < 6 ), pair_printer(std::cout) );

    // Count the number of elements where the data is equal to 2 from a
    // range of elements of the left map view

    count_if( bm.left.range( 2.3 < _key, _key < 5.4 ),
            bind<int>( second_extractor(), _1 ) == 2 );

    return 0;
}
```

Go to source code

Boost.Foreach

In C++, writing a loop that iterates over a sequence is tedious. We can either use iterators, which requires a considerable amount of boiler-plate, or we can use the std::for_each() algorithm and move our loop body into a predicate, which requires no less boiler-plate and forces us to move our logic far from where it will be used. In contrast, some other languages, like Perl, provide a dedicated "foreach" construct that automates this process. BOOST_FOREACH is just such a construct for C++. It iterates over sequences for us, freeing us from having to deal directly with iterators or write predicates.

You can use BOOST_FOREACH macro with Boost.Bimap views. The generated code will be as efficient as a std::for_each iteration. Here are some examples:

```
typedef bimap< std::string, list_of<int> > bm_type;

bm_type bm;
bm.insert( bm_type::value_type("1", 1) );
bm.insert( bm_type::value_type("2", 2) );
bm.insert( bm_type::value_type("3", 4) );
bm.insert( bm_type::value_type("4", 2) );

BOOST_FOREACH( bm_type::left_reference p, bm.left )
{
    ++p.second;  ❶
}

BOOST_FOREACH( bm_type::right_const_reference p, bm.right )
{
    std::cout << p.first << "-->" << p.second << std::endl;
}
```

❶   We can modify the right element because we have use a mutable collection type in the right side.

You can use it directly with ranges too:

```
BOOST_FOREACH( bm_type::left_reference p,
            ( bm.left.range( std::string("1") <= _key, _key < std::string("3") ) ))
{
    ++p.second;
}

BOOST_FOREACH( bm_type::left_const_reference p,
            ( bm.left.range( std::string("1") <= _key, _key < std::string("3") ) ))
{
    std::cout << p.first << "-->" << p.second << std::endl;
}
```

Go to source code

Boost.Typeof

Once C++0x is out we are going to be able to write code like:

```
auto iter = bm.by<name>().find("john");
```

instead of the more verbose

```
bm_type::map_by<name>::iterator iter = bm.by<name>().find("john");
```

Boost.Typeof defines a macro BOOST_AUTO that can be used as a library solution to the auto keyword while we wait for the next standard.

If we have

```
typedef bimap< tagged<std::string,name>, tagged<int,number> > bm_type;
bm_type bm;
bm.insert( bm_type::value_type("one"  ,1) );
bm.insert( bm_type::value_type("two"  ,2) );
```

The following code snippet

```
for( bm_type::map_by<name>::iterator iter = bm.by<name>().begin();
     iter!=bm.by<name>().end(); ++iter)
{
    std::cout << iter->first << " --> " << iter->second << std::endl;
}

bm_type::map_by<number>::iterator iter = bm.by<number>().find(2);
std::cout << "2: " << iter->get<name>();
```

can be rewrited as

```
for( BOOST_AUTO(iter, bm.by<name>().begin()); iter!=bm.by<name>().end(); ++iter)
{
    std::cout << iter->first << " --> " << iter->second << std::endl;
}

BOOST_AUTO( iter, bm.by<number>().find(2) );
std::cout << "2: " << iter->get<name>();
```

Go to source code

Boost.Xpressive

Using Boost.Xpressive we can parse a file and insert the relations in a bimap in the same step. It is just amazing the power of four lines of code. Here is an example (it is just beatifull)

```
typedef bimap< std::string, int > bm_type;
bm_type bm;

std::string rel_str("one <--> 1    two <--> 2     three <--> 3");

sregex rel = ( (s1= +_w) >> " <--> " >> (s2= +_d) )
[
    xp::ref(bm)->*insert( xp::construct<bm_type::value_type>(s1, as<int>(s2)) )
];

sregex relations = rel >> *(+_s >> rel);

regex_match(rel_str, relations);

assert( bm.size() == 3 );
```

Go to source code

Boost.Property_map

The Boost Property Map Library consists mainly of interface specifications in the form of concepts (similar to the iterator concepts in the STL). These interface specifications are intended for use by implementers of generic libraries in communicating requirements on template parameters to their users. In particular, the Boost Property Map concepts define a general purpose interface for mapping key objects to corresponding value objects, thereby hiding the details of how the mapping is implemented from algorithms.

The need for the property map interface came from the Boost Graph Library (BGL), which contains many examples of algorithms that use the property map concepts to specify their interface. For an example, note the ColorMap template parameter of the breadth_first_search. In addition, the BGL contains many examples of concrete types that implement the property map interface. The adjacency_list class implements property maps for accessing objects (properties) that are attached to vertices and edges of the graph.

The counterparts of two of the views of Boost.Bimap map, the `set` and `unordered_set`, are read-write property maps. In order to use these, you need to include one of the following headers:

```
#include <boost/bimap/property_map/set_support.hpp>
#include <boost/bimap/property_map/unordered_set_support.hpp>
```

The following is adapted from the example in the Boost.PropertyMap documentation.

Go to source code

```cpp
template <typename AddressMap>
void foo(AddressMap & address_map)
{
    typedef typename boost::property_traits<AddressMap>::value_type value_type;
    typedef typename boost::property_traits<AddressMap>::key_type key_type;

    value_type address;
    key_type fred = "Fred";
    std::cout << get(address_map, fred);
}

int main()
{
    typedef bimap<std::string, multiset_of<std::string> > Name2Address;
    typedef Name2Address::value_type location;

    Name2Address name2address;
    name2address.insert( location("Fred", "710 West 13th Street") );
    name2address.insert( location( "Joe", "710 West 13th Street") );

    foo( name2address.left );

    return 0;
}
```

Dependencies

Boost.Bimap is built on top of several Boost libraries. The rationale behind this decision is keeping the Boost code base small by reusing existent code. The libraries used are well-established and have been tested extensively, making this library easy to port since all the hard work has already been done. The glue that holds everything together is Boost.MPL. Clearly Boost.MultiIndex is the heart of this library.

Table 4. Boost Libraries needed by Boost.Bimap

Name	Description	author
Boost.MultiIndex	Containers with multiple STL-compatible access interfaces	Joaquín M López Muñoz
Boost.MPL	Template metaprogramming framework of compile-time algorithms, sequences and metafunction classes	Aleksey Gurtovoy
Boost.TypeTraits	Templates for fundamental properties of types.	John Maddock, Steve Cleary
Boost.enable_if	Selective inclusion of function template overloads	Jaakko Järvi, Jeremiah Willcock, Andrew Lumsdaine
Boost.Iterators	Iterator construction framework, adaptors, concepts, and more.	Dave Abrahams, Jeremy Siek, Thomas Witt
Boost.call_traits	Defines types for passing parameters.	John Maddock, Howard Hinnant
Boost.StaticAssert	Static assertions (compile time assertions).	John Maddock

Table 5. Optional Boost Libraries

Name	Description	author	Purpose
Boost.Serialization	Serialization for persistence and marshalling	Robert Ramey	Serialization support for bimap containers and iterators
Boost.Assign	Filling containers with constant or generated data has never been easier	Thorsten Ottosen	Help to fill a bimap or views of it
Boost.Hash	A TR1 hash function object that can be extended to hash user defined types	Daniel James	Default hashing function
Boost.Lambda	Define small unnamed function objects at the actual call site, and more	from Jaakko Järvi, Gary Powell	Functors for modify, range, lower_bound and upper_bound
Boost.Range	A new infrastructure for generic algorithms that builds on top of the new iterator concepts	Thorsten Ottosen	Range based algorithms
Boost.PropertyMap	Concepts defining interfaces which map key objects to value objects	Jeremy Siek	Integration with BGL

Table 6. Additional Boost Libraries needed to run the test-suite

Name	Description	author
Boost.Test	Support for simple program testing, full unit testing, and for program execution monitoring.	Gennadiy Rozental

Reference

Headers

The following are the interface headers of Boost.Bimap:

Convenience

- "boost/bimap.hpp" *(includes "boost/bimap/bimap.hpp" and imports the bimap class to boost namespace)*

Container

- "boost/bimap/bimap.hpp" *(includes "boost/bimap/set_of.hpp" and "boost/bimap/unconstrained_set_of.hpp")*

Set Types

- "boost/bimap/set_of.hpp"

- "boost/bimap/multiset_of.hpp"

- "boost/bimap/unordered_set_of.hpp"

- "boost/bimap/unordered_multiset_of.hpp"

- "boost/bimap/list_of.hpp"

- "boost/bimap/vector_of.hpp"

- "boost/bimap/unconstrained_set_of.hpp"

Boost Integration

- "boost/bimap/support/lambda.hpp"

- "boost/bimap/property_map/set_support.hpp"

- "boost/bimap/property_map/unordered_set_support.hpp"

A program using Boost.Bimap must therefore include "boost/bimap/bimap.hpp" and the headers defining the collection types to be used.

Additional headers allow the integration of Boost.Bimap with other boost libraries, like Boost.Lambda and Boost.Property_map.

In order to use the serialization capabilities of Boost.Bimap, the appropriate Boost.Serialization library module must be linked. Other than that, Boost.Bimap is a header-only library, requiring no additional object modules.

Bimap Reference

View concepts

`bimap` instantiations comprise two side views and an view of the relation specified at compile time. Each view allows read-write access to the elements contained in a definite manner, mathing an STL container signature.

Views are not isolated objects and so cannot be constructed on their own; rather they are an integral part of a `bimap`. The name of the view class implementation proper is never directly exposed to the user, who has access only to the associated view type specifier.

Insertion and deletion of elements are always performed through the appropriate interface of any of the three views of the `bimap`; these operations do, however, have an impact on all other views as well: for instance, insertion through a given view may fail because

there exists another view that forbids the operation in order to preserve its invariant (such as uniqueness of elements). The global operations performed jointly in the any view can be reduced to six primitives:

- copying

- insertion of an element

- hinted insertion, where a pre-existing element is suggested in order to improve the efficiency of the operation

- deletion of an element

- replacement of the value of an element, which may trigger the rearrangement of this element in one or more views, or may forbid the replacement

- modification of an element, and its subsequent rearrangement/banning by the various views

The last two primitives deserve some further explanation: in order to guarantee the invariants associated to each view (e.g. some definite ordering) elements of a `bimap` are not mutable. To overcome this restriction, the views expose member functions for updating and modifying, which allows for the mutation of elements in a controlled fashion.

Complexity signature

Some member functions of a view interface are implemented by global primitives from the above list. The complexity of these operations thus depends on all views of a given `bimap`, not just the currently used view.

In order to establish complexity estimates, a view is characterised by its complexity signature, consisting of the following associated functions on the number of elements:

- `c(n)`: copying

- `i(n)`: insertion

- `h(n)`: hinted insertion

- `d(n)`: deletion

- `r(n)`: replacement

- `m(n)`: modifying

If the collection type of the relation is `left_based` or `right_based`, and we use an `l` subscript to denote the left view and an `r` for the right view, then the insertion of an element in such a container is of complexity `O(i_l(n)+i_r(n))`, where n is the number of elements. If the collection type of relation is not side-based, then there is an additional term to add that is contributed by the collection type of relation view. Using `a` to denote the above view, the complexity of insertion will now be `O(i_l(n)+i_r(n)+i_a(n))`. To abbreviate the notation, we adopt the following definitions:

- `C(n) = c_l(n) + c_r(n) [ + c_a(n) ]`

- `I(n) = i_l(n) + i_r(n) [ + i_a(n) ]`

- `H(n) = h_l(n) + h_r(n) [ + h_a(n) ]`

- `D(n) = d_l(n) + d_r(n) [ + d_a(n) ]`

- `R(n) = r_l(n) + r_r(n) [ + r_a(n) ]`

- `M(n) = m_l(n) + m_r(n) [ + m_a(n) ]`

Set type specification

Set type specifiers are passed as instantiation arguments to `bimap` and provide the information needed to incorporate the corresponding views. Currently, Boost.Bimap provides the collection type specifiers. The *side collection type* specifiers define the constraints of the two map views of the bimap. The *collection type of relation* specifier defines the main set view constraints. If `left_based` (the default parameter) or `right_based` is used, then the collection type of relation will be based on the left or right collection type correspondingly.

Side collection type	Collection type of relation	Include
`set_of`	`set_of_relation`	`boost/bimap/set_of.hpp`
`multiset_of`	`multiset_of_relation`	`boost/bimap/multiset_of.hpp`
`unordered_set_of`	`unordered_set_of_relation`	`boost/bimap/unordered_set_of.hpp`
`unordered_multiset_of`	`unordered_multiset_of_relation`	`boost/bimap/unordered_multiset_of.hpp`
`list_of`	`list_of_relation`	`boost/bimap/list_of.hpp`
`vector_of`	`vector_of_relation`	`boost/bimap/vector_of.hpp`
`unconstrained_set_of`	`unconstrained_set_of_relation`	`boost/bimap/unconstrained_set_of.hpp`
	`left_based`	`boost/bimap/bimap.hpp`
	`right_based`	`boost/bimap/bimap.hpp`

Tags

Tags are just conventional types used as mnemonics for the types stored in a `bimap`. Boost.Bimap uses the tagged idiom to let the user specify this tags.

Header "boost/bimap/bimap.hpp" synopsis

```cpp
namespace boost {
namespace bimaps {

template< class Type, typename Tag >
struct tagged;

// bimap template class

template
<
    class LeftCollectionType, class RightCollectionType,

    class AdditionalParameter_1 = detail::not_specified,
    class AdditionalParameter_2 = detail::not_specified
>
class bimap - implementation defined { : public SetView } -
{
    public:

    // Metadata

    typedef -unspecified- left_tag;
    typedef -unspecified- left_map;

    typedef -unspecified- right_tag;
    typedef -unspecified- right_map;

    // Shortcuts
    // typedef -side-_map::-type- -side-_-type-;

    typedef -unspecified- info_type;

    // Map views

     left_map  left;
    right_map right;

    // Constructors

    bimap();

    template< class InputIterator >
    bimap(InputIterator first,InputIterator last);

    bimap(const bimap &);

    bimap& operator=(const bimap& b);

    // Projection of iterators

    template< class IteratorType >
    left_iterator project_left(IteratorType iter);

    template< class IteratorType >
    left_const_iterator project_left(IteratorType iter) const;

    template< class IteratorType >
    right_iterator project_right(IteratorType iter);

    template< class IteratorType >
    right_const_iterator project_right(IteratorType iter) const;
```

56

```
    template< class IteratorType >
    iterator project_up(IteratorType iter);

    template< class IteratorType >
    const_iterator project_up(IteratorType iter) const;

    // Support for tags

    template< class Tag >
    struct map_by;

    template< class Tag >
    map_by<Tag>::type by();

    template< class Tag >
    const map_by<Tag>::type & by() const;

    template< class Tag, class IteratorType >
    map_by<Tag>::iterator project(IteratorType iter);

    template< class Tag, class IteratorType >
    map_by<Tag>::const_iterator project(IteratorType iter) const

};


} // namespace bimap
} // namespace boost
```

Class template bimap

This is the main component of Boost.Bimap.

Complexity

In the descriptions of the operations of `bimap`, we adopt the scheme outlined in the complexity signature section.

Instantiation types

`bimap` is instantiated with the following types:

1. LeftCollectionType and RightCollectionType are collection type specifications optionally tagged, or any type optionally tagged, in which case that side acts as a set.

2. AdditionalParameter_{1/2} can be any ordered subset of:

   - CollectionTypeOfRelation specification

   - Allocator

Nested types

```
left_tag, right_tag
```

>   Tags for each side of the bimap. If the user has not specified any tag the tags default to `member_at::left` and `member_at::right`.

```
left_key_type, right_key_type
```

Key type of each side. In a `bimap<A,B>` `left_key_type` is `A` and `right_key_type` is `B`. If there are tags, it is better to use: `Bimap::map_by<Tag>::key_type`.

```
left_data_type, right_data_type
```

Data type of each side. In a bimap<A,B> left_key_type is B and right_key_type is A. If there are tags, it is better to use: `Bimap::map_by<Tag>::data_type`.

```
left_value_type, right_value_type
```

Value type used for the views. If there are tags, it is better to use: `Bimap::map_by<Tag>::value_type`.

```
left_iterator, right_iterator
left_const_iterator, right_const_iterator
```

Iterators of the views. If there are tags, it is better to use: `Bimap::map_by<Tag>::iterator` and `Bimap::map_by<Tag>::const_iterator`

```
left_map, right_map
```

Map view type of each side. If there are tags, it is better to use: `Bimap::map_by<Tag>::type`.

Constructors, copy and assignment

```
bimap();
```

- **Effects:** Constructs an empty `bimap`.

- **Complexity:** Constant.

```cpp
template<typename InputIterator>
bimap(InputIterator first,InputIterator last);
```

- **Requires:** `InputIterator` is a model of Input Iterator over elements of type `relation` or a type convertible to `relation`. last is reachable from `first`.

- **Effects:** Constructs an empty `bimap` and fills it with the elements in the range `[first,last)`. Insertion of each element may or may not succeed depending on acceptance by the collection types of the `bimap`.

- **Complexity:** O(m*H(m)), where m is the number of elements in `[first,last)`.

```cpp
bimap(const bimap & x);
```

- **Effects:** Constructs a copy of x, copying its elements as well as its internal objects (key extractors, comparison objects, allocator.)

- **Postconditions:** `*this == x`. The order of the views of the `bimap` is preserved as well.

- **Complexity:** O(x.size()*log(x.size()) + C(x.size()))

```
~bimap()
```

- **Effects:** Destroys the `bimap` and all the elements contained. The order in which the elements are destroyed is not specified.

- **Complexity:** O(n).

```
bimap& operator=(const bimap& x);
```

- **Effects:** Replaces the elements and internal objects of the `bimap` with copies from x.

- **Postconditions:** `*this==x`. The order on the views of the `bimap` is preserved as well.

- **Returns:** `*this`.

- **Complexity:** O(n + x.size()*log(x.size()) + C(x.size())).

- **Exception safety:** Strong, provided the copy and assignment operations of the types of `ctor_args_list` do not throw.

Projection operations

Given a `bimap` with views v1 and v2, we say than an v1-iterator it1 and an v2-iterator it2 are equivalent if:

- `it1 == i1.end()` AND `it2 == i2.end()`,

- OR `it1` and `it2` point to the same element.

```
template< class IteratorType >
left_iterator project_left(IteratorType iter);

template< class IteratorType >
left_const_iterator project_left(IteratorType iter) const;
```

- **Requires:** `IteratorType` is a bimap view iterator. it is a valid iterator of some view of `*this` (i.e. does not refer to some other `bimap`.)

- **Effects:** Returns a left map view iterator equivalent to `it`.

- **Complexity:** Constant.

- **Exception safety:** nothrow.

```
template< class IteratorType >
right_iterator project_right(IteratorType iter);

template< class IteratorType >
right_const_iterator project_right(IteratorType iter) const;
```

- **Requires:** `IteratorType` is a bimap view iterator. it is a valid iterator of some view of `*this` (i.e. does not refer to some other `bimap`.)

- **Effects:** Returns a right map view iterator equivalent to `it`.

- **Complexity:** Constant.

- **Exception safety:** nothrow.

```
template< class IteratorType >
iterator project_up(IteratorType iter);

template< class IteratorType >
const_iterator project_up(IteratorType iter) const;
```

- **Requires:** `IteratorType` is a bimap view iterator. it is a valid iterator of some view of `*this` (i.e. does not refer to some other `bimap`.)

- **Effects:** Returns a collection of relations view iterator equivalent to `it`.

- **Complexity:** Constant.

- **Exception safety:** nothrow.

Support for user defined names

```
template< class Tag >
struct map_by;
```

- `map_by<Tag>::type` yields the type of the map view tagged with `Tag`. `map_by<Tag>::`-*type name*- is the same as `map_by<Tag>::type::`-*type name*-.

- **Requires:** `Tag` is a valid user defined name of the bimap.

```
template< class Tag >
map_by<Tag>::type by();

template< class Tag >
const map_by<Tag>::type & by() const;
```

- **Requires:** `Tag` is a valid user defined name of the bimap.

- **Effects:** Returns a reference to the map view tagged with `Tag` held by `*this`.

- **Complexity:** Constant.

- **Exception safety:** nothrow.

```
template< class Tag, class IteratorType >
map_by<Tag>::iterator project(IteratorType iter);

template< class Tag, class IteratorType >
map_by<Tag>::const_iterator project(IteratorType iter) const
```

- **Requires:** `Tag` is a valid user defined name of the bimap. `IteratorType` is a bimap view iterator. it is a valid iterator of some view of `*this` (i.e. does not refer to some other `bimap`.)

- **Effects:** Returns a reference to the map view tagged with `Tag` held by `*this`.

- **Complexity:** Constant.

- **Exception safety:** nothrow.

Serialization

A `bimap` can be archived and retrieved by means of **Boost.Serialization**. Boost.Bimap does not expose a public serialisation interface, as this is provided by Boost.Serialization itself. Both regular and XML archives are supported.

Each of the set specifications comprising a given `bimap` contributes its own preconditions as well as guarantees on the retrieved containers. In describing these, the following concepts are used. A type `T` is *serializable* (resp. XML-serializable) if any object of type `T` can be saved to an output archive (XML archive) and later retrieved from an input archive (XML archive) associated to the same storage. If x' of type `T` is loaded from the serialization information saved from another object x, we say that x' is a *restored copy* of x. Given a Binary Predicate `Pred` over (`T`, `T`), and objects p and q of type `Pred`, we say that q is *serialization-compatible* with p if

- `p(x,y) == q(x',y')`

for every x and y of type T and x' and y' being restored copies of x and y, respectively.

> **Operation:** saving of a `bimap` `b` to an output archive (XML archive) ar.

- **Requires:** Value is serializable (XML-serializable). Additionally, each of the views of b can impose other requirements.

- **Exception safety:** Strong with respect to `b`. If an exception is thrown, ar may be left in an inconsistent state.

> **Operation:** loading of a `bimap` m' from an input archive (XML archive) ar.

- **Requires:** Value is serializable (XML-serializable). Additionally, each of the views of `b'` can impose other requirements.

- **Exception safety:** Basic. If an exception is thrown, ar may be left in an inconsistent state.

set_of Reference

Header "boost/bimap/set_of.hpp" synopsis

```cpp
namespace boost {
namespace bimaps {


template
<
    class KeyType,
    class KeyCompare = std::less< KeyType >
>
struct set_of;


template
<
    class KeyCompare = std::less< _relation >
>
struct set_of_relation;


} // namespace bimap
} // namespace boost
```

Header "boost/bimap/multiset_of.hpp" synopsis

```
namespace boost {
namespace bimaps {


template
<
    class KeyType,
    class KeyCompare = std::less< KeyType >
>
struct multiset_of;


template
<
    class KeyCompare = std::less< _relation >
>
struct multiset_of_relation;


} // namespace bimap
} // namespace boost
```

Collection type specifiers set_of and multiset_of

These collection type specifiers allow for insertion of sets disallowing or allowing duplicate elements, respectively. The syntaxes of `set_of` and `multiset_of` coincide, so they are described together.

[multi]set_of Views

A [multi]set_of set view is a std::[multi]set signature-compatible interface to the underlying heap of elements contained in a `bimap`.

There are two variants: set_of, which does not allow duplicate elements (with respect to its associated comparison predicate) and multiset_of, which does accept those duplicates. The interface of these two variants is largely the same, so they are documented together with their differences explicitly noted where they exist.

If you look the bimap from a side, you will use a map view, and if you look at it as a whole, you will be using a set view.

```cpp
namespace boost {
namespace bimaps {
namespace views {

template< -implementation defined parameter list- >
class -implementation defined view name-
{
    public:

    typedef -unspecified- key_type;
    typedef -unspecified- value_type;
    typedef -unspecified- key_compare;
    typedef -unspecified- value_compare;
    typedef -unspecified- allocator_type;
    typedef -unspecified- reference;
    typedef -unspecified- const_reference;
    typedef -unspecified- iterator;
    typedef -unspecified- const_iterator;
    typedef -unspecified- size_type;
    typedef -unspecified- difference_type;
    typedef -unspecified- pointer;
    typedef -unspecified- const_pointer;
    typedef -unspecified- reverse_iterator;
    typedef -unspecified- const_reverse_iterator;

    typedef -unspecified- info_type;

    this_type & operator=(const this_type & x);

    allocator_type get_allocator() const;

    // iterators

    iterator               begin();
    const_iterator         begin() const;

    iterator               end();
    const_iterator         end() const;

    reverse_iterator       rbegin();
    const_reverse_iterator rbegin() const;

    reverse_iterator       rend();
    const_reverse_iterator rend() const;

    // capacity

    bool      empty() const;

    size_type size() const;

    size_type max_size() const;

    // modifiers

    std::pair<iterator,bool> insert(const value_type & x);

    iterator insert(iterator position, const value_type & x);

    template< class InputIterator>
    void insert(InputIterator first,  InputIterator last);

    iterator erase(iterator position);
```

```cpp
template< class CompatibleKey >
size_type erase(const CompatibleKey & x);

iterator erase(iterator first,  iterator last);

bool replace(iterator position, const value_type& x);

// Only in map views
// {

  template< class CompatibleKey >
  bool replace_key(iterator position, const CompatibleKey & x);

  template< class CompatibleData >
  bool replace_data(iterator position, const CompatibleData & x);

  template< class KeyModifier >
  bool modify_key(iterator position, KeyModifier mod);

  template< class DataModifier >
  bool modify_data(iterator position, DataModifier mod);

// }

void swap(this_type & x);

void clear();

// observers

key_compare     key_comp() const;

value_compare   value_comp() const;

// set operations

template< class CompatibleKey >
iterator find(const CompatibleKey & x);

template< class CompatibleKey >
const_iterator find(const CompatibleKey & x) const;


template< class CompatibleKey >
size_type count(const CompatibleKey & x) const;


template< class CompatibleKey >
iterator lower_bound(const CompatibleKey & x);

template< class CompatibleKey >
const_iterator lower_bound(const CompatibleKey & x) const;


template< class CompatibleKey >
iterator upper_bound(const CompatibleKey & x);

template< class CompatibleKey >
const_iterator upper_bound(const CompatibleKey & x) const;


template< class CompatibleKey >
```

```cpp
    std::pair<iterator,iterator>
        equal_range(const CompatibleKey & x);

    template< class CompatibleKey >
    std::pair<const_iterator,const_iterator>
        equal_range(const CompatibleKey & x) const;

    // Only in maps views
    // {

    template< class LowerBounder, class UpperBounder>
    std::pair<iterator,iterator> range(
        LowerBounder lower, UpperBounder upper);

    template< class LowerBounder, class UpperBounder>
    std::pair<const_iterator,const_iterator> range(
        LowerBounder lower, UpperBounder upper) const;

    typedef -unspecified- mapped_type;
    typedef -unspecified- data_type; // Equal to mapped_type

      // Only in for `set_of` collection type
      // {

      template< class CompatibleKey >
      const mapped_type & at(const CompatibleKey & k) const;

        // Only if the other collection type is mutable
        // {

        template< class CompatibleKey >
        mapped_type & operator[](const CompatibleKey & k);

        template< class CompatibleKey >
        mapped_type & at(const CompatibleKey & k);

        // }

        // Only if info_hook is used
        // {

        template< class CompatibleKey >
        info_type & info_at(const CompatibleKey & k);

        template< class CompatibleKey >
        const info_type & info_at(const CompatibleKey & k) const;

        // }

      // }

    // }
};

// view comparison

bool operator==(const this_type & v1, const this_type & v2 );
bool operator< (const this_type & v1, const this_type & v2 );
bool operator!=(const this_type & v1, const this_type & v2 );
bool operator> (const this_type & v1, const this_type & v2 );
```

```
bool operator>=(const this_type & v1, const this_type & v2 );
bool operator<=(const this_type & v1, const this_type & v2 );

} // namespace views
} // namespace bimap
} // namespace boost
```

In the case of a `bimap< {multi}set_of<Left>, ... >`

In the set view:

```
typedef signature-compatible with relation<       Left, ... > key_type;
typedef signature-compatible with relation< const Left, ... > value_type;
```

In the left map view:

```
typedef  Left  key_type;
typedef  ...   mapped_type;

typedef signature-compatible with std::pair< const Left, ... > value_type;
```

In the right map view:

```
typedef  ...   key_type;
typedef  Left mapped_type;

typedef signature-compatible with std::pair< ... ,const Left > value_type;
```

Complexity signature

Here and in the descriptions of operations of this view, we adopt the scheme outlined in the complexity signature section. The complexity signature of [multi]set_of view is:

- copying: `c(n) = n * log(n)`,

- insertion: `i(n) = log(n)`,

- hinted insertion: `h(n) = 1` (constant) if the hint element precedes the point of insertion, `h(n) = log(n)` otherwise,

- deletion: `d(n) = 1` (amortized constant),

- replacement: `r(n) = 1` (constant) if the element position does not change, `r(n) = log(n)` otherwise,

- modifying: `m(n) = 1` (constant) if the element position does not change, `m(n) = log(n)` otherwise.

Instantiation types

Set views are instantiated internally to a `bimap`. Instantiations are dependent on the following types:

- `Value` from the set specifier,

- `Allocator` from `bimap`,

- `Compare` from the set specifier.

`Compare` is a Strict Weak Ordering on elements of `Value`.

Constructors, copy and assignment

Set views do not have public constructors or destructors. Assignment, on the other hand, is provided.

```
this_type & operator=(const this_type & x);
```

- **Effects:** `a = b;` where a and b are the `bimap` objects to which `*this` and x belong, respectively.

- **Returns:** `*this`.

Modifiers

```
std::pair<iterator,bool> insert(const value_type & x);
```

- **Effects:** Inserts `x` into the `bimap` to which the set view belongs if

  - the set view is non-unique OR no other element with equivalent key exists,

  - AND insertion is allowed by the other set specifications the `bimap`.

- **Returns:** The return value is a pair `p`. `p.second` is `true` if and only if insertion took place. On successful insertion, `p.first` points to the element inserted; otherwise, `p.first` points to an element that caused the insertion to be banned. Note that more than one element can be causing insertion not to be allowed.

- **Complexity:** O(I(n)).

- **Exception safety:** Strong.

```
iterator insert(iterator position, const value_type & x);
```

- **Requires:** `position` is a valid iterator of the view.

- **Effects:** `position` is used as a hint to improve the efficiency of the operation. Inserts `x` into the `bimap` to which the view belongs if

  - the set view is non-unique OR no other element with equivalent key exists,

  - AND insertion is allowed by all other views of the `bimap`.

- **Returns:** On successful insertion, an iterator to the newly inserted element. Otherwise, an iterator to an element that caused the insertion to be banned. Note that more than one element can be causing insertion not to be allowed.

- **Complexity:** O(H(n)).

- **Exception safety:** Strong.

```
template< class InputIterator >
void insert(InputIterator first, InputIterator last);
```

- **Requires:** `InputIterator` is a model of Input Iterator over elements of type `value_type` or a type convertible to value_type. `first` and `last` are not iterators into any view of the `bimap` to which this index belongs. `last` is reachable from `first`.

- **Effects:** `iterator hint = end();while( first != last ) hint = insert( hint, *first++ );`

- **Complexity:** O(m*H(n+m)), where m is the number of elements in `[first, last)`.

- **Exception safety:** Basic.

```
iterator erase(iterator position);
```

- **Requires:** `position` is a valid dereferenceable iterator if the set view.

- **Effects:** Deletes the element pointed to by `position`.

- **Returns:** An iterator pointing to the element immediately following the one that was deleted, or `end()` if no such element exists.

- **Complexity:** O(D(n)).

- **Exception safety:** nothrow.

```
template< class CompatibleKey >
size_type erase(const CompatibleKey & x);
```

- **Requires:** `CompatibleKey` is a compatible key of `key_compare`.

- **Effects:** Deletes the elements with key equivalent to `x`.

- **Returns:** Number of elements deleted.

- **Complexity:** O(log(n) + m*D(n)), where m is the number of elements deleted.

- **Exception safety:** Basic.

```
iterator erase(iterator first, iterator last);
```

- **Requires:** `[first,last)` is a valid range of the view.

- **Effects:** Deletes the elements in `[first,last)`.

- **Returns:** last.

- **Complexity:** O(log(n) + m*D(n)), where m is the number of elements in `[first,last)`.

- **Exception safety:** nothrow.

```
bool replace(iterator position, const value_type& x);
```

- **Requires:** `position` is a valid dereferenceable iterator of the set view.

- **Effects:** Assigns the value `x` to the element pointed to by `position` into the `bimap` to which the set view belongs if, for the value `x`

  - the set view is non-unique OR no other element with equivalent key exists (except possibly `*position`),

  - AND replacing is allowed by all other views of the `bimap`.

- **Postconditions:** Validity of position is preserved in all cases.

- **Returns:** `true` if the replacement took place, `false` otherwise.

- **Complexity:** O(R(n)).

- **Exception safety:** Strong. If an exception is thrown by some user-provided operation, the `bimap` to which the set view belongs remains in its original state.

```
template< class CompatibleKey >
bool replace_key(iterator position, const CompatibleKey & x);
```

- **Requires:** `position` is a valid dereferenceable iterator of the set view. `CompatibleKey` can be assigned to `key_type`.

- **Effects:** Assigns the value `x` to `e.first`, where `e` is the element pointed to by `position` into the `bimap` to which the set view belongs if,

  - the map view is non-unique OR no other element with equivalent key exists (except possibly `*position`),

  - AND replacing is allowed by all other views of the `bimap`.

- **Postconditions:** Validity of position is preserved in all cases.

- **Returns:** `true` if the replacement took place, `false` otherwise.

- **Complexity:** O(R(n)).

- **Exception safety:** Strong. If an exception is thrown by some user-provided operation, the `bimap` to which the set view belongs remains in its original state.

```
template< class CompatibleData >
bool replace_data(iterator position, const CompatibleData & x);
```

- **Requires:** `position` is a valid dereferenceable iterator of the set view. `CompatibleKey` can be assigned to `mapped_type`.

- **Effects:** Assigns the value `x` to `e.second`, where `e` is the element pointed to by `position` into the `bimap` to which the set view belongs if,

  - the map view is non-unique OR no other element with equivalent key exists (except possibly `*position`),

  - AND replacing is allowed by all other views of the `bimap`.

- **Postconditions:** Validity of position is preserved in all cases.

- **Returns:** `true` if the replacement took place, `false` otherwise.

- **Complexity:** O(R(n)).

- **Exception safety:** Strong. If an exception is thrown by some user-provided operation, the `bimap` to which the set view belongs remains in its original state.

```
template< class KeyModifier >
bool modify_key(iterator position, KeyModifier mod);
```

- **Requires:** `KeyModifier` is a model of Unary Function accepting arguments of type: `key_type&`; `position` is a valid dereferenceable iterator of the view.

- **Effects:** Calls `mod(e.first)` where e is the element pointed to by position and rearranges `*position` into all the views of the `bimap`. If the rearrangement fails, the element is erased. Rearrangement is successful if

  - the map view is non-unique OR no other element with equivalent key exists,

  - AND rearrangement is allowed by all other views of the `bimap`.

- **Postconditions:** Validity of `position` is preserved if the operation succeeds.

- **Returns:** `true` if the operation succeeded, `false` otherwise.

- **Complexity:** O(M(n)).

- **Exception safety:** Basic. If an exception is thrown by some user-provided operation (except possibly mod), then the element pointed to by position is erased.

- **Note:** Only provided for map views.

```
template< class DataModifier >
bool modify_data(iterator position, DataModifier mod);
```

- **Requires:** DataModifier is a model of Unary Function accepting arguments of type: mapped_type&; position is a valid dereferenceable iterator of the view.

- **Effects:** Calls mod(e.second) where e is the element pointed to by position and rearranges *position into all the views of the bimap. If the rearrangement fails, the element is erased. Rearrangement is successful if

  - the oppositte map view is non-unique OR no other element with equivalent key in that view exists,

  - AND rearrangement is allowed by all other views of the bimap.

- **Postconditions:** Validity of position is preserved if the operation succeeds.

- **Returns:** true if the operation succeeded, false otherwise.

- **Complexity:** O(M(n)).

- **Exception safety:** Basic. If an exception is thrown by some user-provided operation (except possibly mod), then the element pointed to by position is erased.

- **Note:** Only provided for map views.

Set operations

[multi]set_of views provide the full lookup functionality required by Sorted Associative Container and Unique Associative Container, namely find, count, lower_bound, upper_bound and equal_range. Additionally, these member functions are templatized to allow for non-standard arguments, so extending the types of search operations allowed.

A type CompatibleKey is said to be a *compatible key* of Compare if (CompatibleKey, Compare) is a compatible extension of Compare. This implies that Compare, as well as being a strict weak ordering, accepts arguments of type CompatibleKey, which usually means it has several overloads of operator().

```
template< class CompatibleKey >
iterator find(const CompatibleKey & x);

template< class CompatibleKey >
const_iterator find(const CompatibleKey & x) const;
```

- **Requires:** CompatibleKey is a compatible key of key_compare.

- **Effects:** Returns a pointer to an element whose key is equivalent to x, or end() if such an element does not exist.

- **Complexity:** O(log(n)).

```
template< class CompatibleKey >
size_type count(const key_type & x) const;
```

- **Requires:** CompatibleKey is a compatible key of key_compare.

- **Effects:** Returns the number of elements with key equivalent to x.

- **Complexity:** O(log(n) + count(x)).

```
template< class CompatibleKey >
iterator lower_bound(const key_type & x);

template< class CompatibleKey >
const_iterator lower_bound(const key_type & x) const;
```

- **Requires:** CompatibleKey is a compatible key of key_compare.

- **Effects:** Returns an iterator pointing to the first element with key not less than x, or end() if such an element does not exist.

- **Complexity:** O(log(n)).

```
template< class CompatibleKey >
iterator upper_bound(const key_type & x);

template< class CompatibleKey >
const_iterator upper_bound(const key_type & x) const;
```

- **Requires:** CompatibleKey is a compatible key of key_compare.

- **Effects:** Returns an iterator pointing to the first element with key greater than x, or end() if such an element does not exist.

- **Complexity:** O(log(n)).

```
template< class CompatibleKey >
std::pair<iterator,iterator>
    equal_range(const key_type & x);

template< class CompatibleKey >
std::pair<const_iterator,const_iterator>
    equal_range(const key_type & x) const;
```

- **Requires:** CompatibleKey is a compatible key of key_compare.

- **Effects:** Equivalent to make_pair(lower_bound(x),upper_bound(x)).

- **Complexity:** O(log(n)).

Range operations

The member function range is not defined for sorted associative containers, but [multi]set_of map views provide it as a convenient utility. A range or interval is defined by two conditions for the lower and upper bounds, which are modelled after the following concepts.

Consider a Strict Weak Ordering Compare over values of type Key. A type LowerBounder is said to be a lower bounder of Compare if

- LowerBounder is a Predicate over Key,

- if lower(k1) and !comp(k2,k1) then lower(k2),

for every lower of type LowerBounder, comp of type Compare, and k1, k2 of type Key. Similarly, an upper bounder is a type UpperBounder such that

- UpperBounder is a Predicate over Key,

- if upper(k1) and !comp(k1,k2) then upper(k2),

for every `upper` of type `UpperBounder`, `comp` of type `Compare`, and `k1`, `k2` of type `Key`.

```
template< class LowerBounder, class UpperBounder>
std::pair<const_iterator,const_iterator> range(
    LowerBounder lower, UpperBounder upper) const;
```

- **Requires:** `LowerBounder` and `UpperBounder` are a lower and upper bounder of `key_compare`, respectively.

- **Effects:** Returns a pair of iterators pointing to the beginning and one past the end of the subsequence of elements satisfying lower and upper simultaneously. If no such elements exist, the iterators both point to the first element satisfying lower, or else are equal to `end()` if this latter element does not exist.

- **Complexity:** O(log(n)).

- **Variants:** In place of lower or upper (or both), the singular value `boost::bimap::unbounded` can be provided. This acts as a predicate which all values of type `key_type` satisfy.

- **Note:** Only provided for map views.

at(), info_at() and operator[] - set_of only

```
template< class CompatibleKey >
const mapped_type & at(const CompatibleKey & k) const;
```

- **Requires:** `CompatibleKey` is a compatible key of `key_compare`.

- **Effects:** Returns the `mapped_type` reference that is associated with k, or throws `std::out_of_range` if such key does not exist.

- **Complexity:** O(log(n)).

- **Note:** Only provided when `set_of` is used.

The symmetry of bimap imposes some constraints on `operator[]` and the non constant version of at() that are not found in `std::maps`. Tey are only provided if the other collection type is mutable (`list_of`, `vector_of` and `unconstrained_set_of`).

```
template< class CompatibleKey >
mapped_type & operator[](const CompatibleKey & k);
```

- **Requires:** `CompatibleKey` is a compatible key of `key_compare`.

- **Effects:** `return insert(value_type(k,mapped_type()))->second;`

- **Complexity:** O(log(n)).

- **Note:** Only provided when `set_of` is used and the other collection type is mutable.

```
template< class CompatibleKey >
mapped_type & at(const CompatibleKey & k);
```

- **Requires:** `CompatibleKey` is a compatible key of `key_compare`.

- **Effects:** Returns the `mapped_type` reference that is associated with k, or throws `std::out_of_range` if such key does not exist.

- **Complexity:** O(log(n)).

- **Note:** Only provided when `set_of` is used and the other collection type is mutable.

```
template< class CompatibleKey >
info_type & info_at(const CompatibleKey & k);

template< class CompatibleKey >
const info_type & info_at(const CompatibleKey & k) const;
```

- **Requires:** `CompatibleKey` is a compatible key of `key_compare`.

- **Effects:** Returns the `info_type` reference that is associated with `k`, or throws `std::out_of_range` if such key does not exist.

- **Complexity:** O(log(n)).

- **Note:** Only provided when `set_of` and `info_hook` are used

Serialization

Views cannot be serialized on their own, but only as part of the `bimap` into which they are embedded. In describing the additional preconditions and guarantees associated to `[multi]set_of` views with respect to serialization of their embedding containers, we use the concepts defined in the `bimap` serialization section.

> **Operation:** saving of a `bimap` m to an output archive (XML archive) ar.

- **Requires:** No additional requirements to those imposed by the container.

> **Operation:** loading of a `bimap` m' from an input archive (XML archive) ar.

- **Requires:** In addition to the general requirements, `value_comp()` must be serialization-compatible with `m.get<i>().value_comp()`, where i is the position of the ordered view in the container.

- **Postconditions:** On successful loading, each of the elements of `[begin(), end())` is a restored copy of the corresponding element in `[m.get<i>().begin(), m.get<i>().end())`.

> **Operation:** saving of an iterator or `const_iterator` it to an output archive (XML archive) ar.

- **Requires:** `it` is a valid iterator of the view. The associated `bimap` has been previously saved.

> **Operation:** loading of an `iterator` or `const_iterator` it' from an input archive (XML archive) ar.

- **Postconditions:** On successful loading, if it was dereferenceable then `*it'` is the restored copy of `*it`, otherwise `it' == end()`.

- **Note:** It is allowed that it be a `const_iterator` and the restored `it'` an iterator, or viceversa.

unordered_set_of Reference

Header "boost/bimap/unordered_set_of.hpp" synopsis

```
namespace boost {
namespace bimaps {


template
<
    class KeyType,
    class HashFunctor   = hash< KeyType >,
    class EqualKey      = std::equal_to< KeyType >
>
struct unordered_set_of;


template
<
    class HashFunctor   = hash< _relation >,
    class EqualKey      = std::equal_to< _relation >
>
struct unordered_set_of_relation;


} // namespace bimap
} // namespace boost
```

Header "boost/bimap/unordered_multiset_of.hpp" synopsis

```
namespace boost {
namespace bimaps {


template
<
    class KeyType,
    class HashFunctor   = hash< KeyType >,
    class EqualKey      = std::equal_to< KeyType >
>
struct unordered_multiset_of;


template
<
    class HashFunctor   = hash< _relation >,
    class EqualKey      = std::equal_to< _relation >
>
struct unordered_multiset_of_relation;


} // namespace bimap
} // namespace boost
```

Collection type specifiers unordered_set_of and unordered_multiset_of

These collection types specifiers allow for set views without and with allowance of duplicate elements, respectively. The syntax of `set_of` and `multiset_of` coincide, thus we describe them in a grouped manner.

unordered_[multi]set_of Views

An unordered_[multi]set_of set view is a tr1::unordered[multi]set signature compatible interface to the underlying heap of elements contained in a `bimap`.

The interface and semantics of unordered_[multi]set_of views are modeled according to the proposal for unordered associative containers given in the C++ Standard Library Technical Report, also known as TR1. An unordered_[multi]set_of view is particularized according to a given `Hash` function object which returns hash values for the keys and a binary predicate `Pred` acting as an equivalence relation on values of Key.

There are two variants: unordered_set_of, which do not allow duplicate elements (with respect to its associated comparison predicate) and unordered_multiset_of, which accept those duplicates. The interface of these two variants is the same to a great extent, so they are documented together with their differences explicitly noted when they exist.

If you look the bimap by a side, you will use a map view and if you looked it as a whole you will be using a set view.

Except where noted, unordered_[multi]set_of views (both unique and non-unique) are models of `Unordered Associative Container`. Validity of iterators and references to elements is preserved in all cases. Occasionally, the exception safety guarantees provided are actually stronger than required by the extension draft. We only provide descriptions of those types and operations that are either not present in the concepts modeled or do not exactly conform to the requirements for unordered associative containers.

```cpp
namespace boost {
namespace bimap {
namespace views {

template< -implementation defined parameter list- >
class -implementation defined view name-
{
    public:

    // types

    typedef -unspecified- key_type;
    typedef -unspecified- value_type;
    typedef -unspecified- key_compare;
    typedef -unspecified- value_compare;
    typedef -unspecified- hasher;
    typedef -unspecified- key_equal;
    typedef -unspecified- allocator_type;
    typedef -unspecified- reference;
    typedef -unspecified- const_reference;
    typedef -unspecified- iterator;
    typedef -unspecified- const_iterator;
    typedef -unspecified- size_type;
    typedef -unspecified- difference_type;
    typedef -unspecified- pointer;
    typedef -unspecified- const_pointer;
    typedef -unspecified- local_iterator;
    typedef -unspecified- const_local_iterator;

    typedef -unspecified- info_type;

    // construct/destroy/copy:

    this_type & operator=(const this_type & x);

    allocator_type get_allocator() const;

    // size and capacity

    bool      empty() const;
```

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

// iterators

iterator       begin();
const_iterator begin() const;
iterator       end();
const_iterator end() const;

// modifiers

std::pair< iterator, bool > insert(const value_type & x);

iterator insert(iterator position, const value_type & x);

template< class InputIterator >
void insert(InputIterator first, InputIterator last);

iterator erase(iterator position);

template< class CompatibleKey >
size_type erase(const CompatibleKey & x);

iterator erase(iterator first, iterator last);

bool replace(iterator position, const value_type & x);

// Only in map views
// {

  typedef -unspecified- mapped_type;
  typedef -unspecified- data_type; // Equal to mapped_type

  template< class CompatibleKey >
  bool replace_key(iterator position, const CompatibleKey & x);

  template< class CompatibleData >
  bool replace_data(iterator position, const CompatibleData & x);

  template< class KeyModifier >
  bool modify_key(iterator position, KeyModifier mod);

  template< class DataModifier >
  bool modify_data(iterator position, DataModifier mod);

// }


void clear();

// observers

key_from_value key_extractor() const;
hasher         hash_function() const;
key_equal      key_eq() const;

// lookup

template< class CompatibleKey >
iterator find(const CompatibleKey & x);

template< class CompatibleKey >
```

```
const_iterator find(const CompatibleKey & x) const;

template< class CompatibleKey >
size_type count(const CompatibleKey & x) const;

template< class CompatibleKey >
std::pair<iterator,iterator>
    equal_range(const CompatibleKey & x);

template< class CompatibleKey >
std::pair<const_iterator,const_iterator>
    equal_range(const CompatibleKey & x) const;

// bucket interface

size_type bucket_count() const;
size_type max_bucket_count() const;
size_type bucket_size(size_type n) const;
size_type bucket(const key_type & k) const;

local_iterator       begin(size_type n);
const_local_iterator begin(size_type n) const;
local_iterator       end(size_type n);
const_local_iterator end(size_type n) const;

// hash policy

float load_factor() const;
float max_load_factor() const;
void  max_load_factor(float z);
void  rehash(size_type n);

// Only in maps views
// {

typedef -unspecified- mapped_type;

  // Only in for `unordered_set_of` collection type
  // {

  template<class CompatibleKey>
  const mapped_type & at(const CompatibleKey & k) const;

    // Only if the other collection type is mutable
    // {

    template<class CompatibleKey>
    mapped_type & operator[](const CompatibleKey & k);

    template<class CompatibleKey>
    mapped_type & at(const CompatibleKey & k);

    // }

    // Only if info_hook is used
    // {

    template< class CompatibleKey >
    info_type & info_at(const CompatibleKey & k);

    template< class CompatibleKey >
    const info_type & info_at(const CompatibleKey & k) const;
```

77

```
      // }

   // }

};

} // namespace views
} // namespace bimap
} // namespace boost
```

In the case of a `bimap< unordered_{multi}set_of<Left>, ... >`

In the set view:

```
typedef signature-compatible with relation<       Left, ... > key_type;
typedef signature-compatible with relation< const Left, ... > value_type;
```

In the left map view:

```
typedef  Left  key_type;
typedef  ...    mapped_type;

typedef signature-compatible with std::pair< const Left, ... > value_type;
```

In the right map view:

```
typedef  ...   key_type;
typedef  Left mapped_type;

typedef signature-compatible with std::pair< ... ,const Left > value_type;
```

Complexity signature

Here and in the descriptions of operations of `unordered_[multi]set_of` views, we adopt the scheme outlined in the complexity signature section. The complexity signature of `unordered_[multi]set_of` view is:

- copying: `c(n) = n * log(n)`,

- insertion: average case `i(n) = 1` (constant), worst case `i(n) = n`,

- hinted insertion: average case `h(n) = 1` (constant), worst case `h(n) = n`,

- deletion: average case `d(n) = 1` (constant), worst case `d(n) = n`,

- replacement:

  - if the new element key is equivalent to the original, `r(n) = 1` (constant),

  - otherwise, average case `r(n) = 1` (constant), worst case `r(n) = n`,

- modifying: average case `m(n) = 1` (constant), worst case `m(n) = n`.

Instantiation types

`unordered_[multi]set_of` views are instantiated internally to `bimap` specified by means of the collection type specifiers and the `bimap` itself. Instantiations are dependent on the following types:

- `Value` from `bimap`,

- `Allocator` from `bimap`,

- `Hash` from the collection type specifier,

- `Pred` from the collection type specifier.

`Hash` is a [Unary Function](#) taking a single argument of type `key_type` and returning a value of type `std::size_t` in the range `[0, std::numeric_limits<std::size_t>::max())`. Pred is a [Binary Predicate](#) inducing an equivalence relation on elements of `key_type`. It is required that the `Hash` object return the same value for keys equivalent under `Pred`.

Nested types

```
iterator
const_iterator
local_iterator
const_local_iterator
```

These types are models of [Forward Iterator](#).

Constructors, copy and assignment

As explained in the concepts section, views do not have public constructors or destructors. Assignment, on the other hand, is provided. Upon construction, `max_load_factor()` is 1.0.

```
this_type & operator=(const this_type & x);
```

- **Effects:** `a = b;` where a and b are the `bimap` objects to which `*this` and x belong, respectively.

- **Returns:** `*this`.

Modifiers

```
std::pair<iterator,bool> insert(const value_type & x);
```

- **Effects:** Inserts `x` into the `bimap` to which the view belongs if

  - the view is non-unique OR no other element with equivalent key exists,

  - AND insertion is allowed by all other views of the `bimap`.

- **Returns:** The return value is a pair p. p.second is `true` if and only if insertion took place. On successful insertion, `p.first` points to the element inserted; otherwise, `p.first` points to an element that caused the insertion to be banned. Note that more than one element can be causing insertion not to be allowed.

- **Complexity:** O(I(n)).

- **Exception safety:** Strong.

```
iterator insert(iterator position, const value_type & x);
```

- **Requires:** `position` is a valid iterator of the view.

- **Effects:** `position` is used as a hint to improve the efficiency of the operation. Inserts `x` into the `bimap` to which the view belongs if

  - the view is non-unique OR no other element with equivalent key exists,

  - AND insertion is allowed by all other views of the `bimap`.

- **Returns:** On successful insertion, an iterator to the newly inserted element. Otherwise, an iterator to an element that caused the insertion to be banned. Note that more than one element can be causing insertion not to be allowed.

- **Complexity:** O(H(n)).

- **Exception safety:** Strong.

```
template< class InputIterator>
void insert(InputIterator first, InputIterator last);
```

- **Requires:** `InputIterator` is a model of [Input Iterator](#) over elements of type `value_type`. `first` and `last` are not iterators into any views of the `bimap` to which this view belongs. `last` is reachable from `first`.

- **Effects:** `iterator hint = end(); while(first != last) hint = insert(hint, *first++);`

- **Complexity:** O(m*H(n+m)), where m is the number of elements in `[first, last)`.

- **Exception safety:** Basic.

```
iterator erase(iterator position);
```

- **Requires:** `position` is a valid dereferenceable `iterator` of the view.

- **Effects:** Deletes the element pointed to by `position`.

- **Returns:** An `iterator` pointing to the element immediately following the one that was deleted, or `end()` if no such element exists.

- **Complexity:** O(D(n)).

- **Exception safety:** nothrow.

```
template< class CompatibleKey >
size_type erase(const CompatibleKey & x);
```

- **Effects:** Deletes the elements with key equivalent to `x`.

- **Returns:** Number of elements deleted.

- **Complexity:** Average case, O(1 + m*D(n)), worst case O(n + m*D(n)), where m is the number of elements deleted.

- **Exception safety:** Basic.

```
iterator erase(iterator first, iterator last);
```

- **Requires:** `[first,last)` is a valid range of the view.

- **Effects:** Deletes the elements in `[first,last)`.

- **Returns:** `last`.

- **Complexity:** O(m*D(n)), where m is the number of elements in `[first,last)`.

- **Exception safety:** nothrow.

```
bool replace(iterator position, const value_type & x);
```

- **Requires:** `position` is a valid dereferenceable `iterator` of the view.

- **Effects:** Assigns the value `x` to the element pointed to by `position` into the `bimap` to which the view belongs if, for the value `x`

  - the view is non-unique OR no other element with equivalent key exists (except possibly `*position`),

  - AND replacing is allowed by all other views of the `bimap`.

- **Postconditions:** Validity of position is preserved in all cases.

- **Returns:** `true` if the replacement took place, `false` otherwise.

- **Complexity:** O(R(n)).

- **Exception safety:** Strong. If an exception is thrown by some user-provided operation the `bimap` to which the view belongs remains in its original state.

```
template< class CompatibleKey >
bool replace_key(iterator position, const CompatibleKey & x);
```

- **Requires:** `position` is a valid dereferenceable iterator of the set view. `CompatibleKey` can be assigned to `key_type`.

- **Effects:** Assigns the value `x` to `e.first`, where `e` is the element pointed to by `position` into the `bimap` to which the set view belongs if,

  - the map view is non-unique OR no other element with equivalent key exists (except possibly `*position`),

  - AND replacing is allowed by all other views of the `bimap`.

- **Postconditions:** Validity of position is preserved in all cases.

- **Returns:** `true` if the replacement took place, `false` otherwise.

- **Complexity:** O(R(n)).

- **Exception safety:** Strong. If an exception is thrown by some user-provided operation, the `bimap` to which the set view belongs remains in its original state.

```
template< class CompatibleData >
bool replace_data(iterator position, const CompatibleData & x);
```

- **Requires:** `position` is a valid dereferenceable iterator of the set view. `CompatibleKey` can be assigned to `mapped_type`.

- **Effects:** Assigns the value `x` to `e.second`, where `e` is the element pointed to by `position` into the `bimap` to which the set view belongs if,

  - the map view is non-unique OR no other element with equivalent key exists (except possibly `*position`),

  - AND replacing is allowed by all other views of the `bimap`.

- **Postconditions:** Validity of position is preserved in all cases.

- **Returns:** `true` if the replacement took place, `false` otherwise.

- **Complexity:** O(R(n)).

- **Exception safety:** Strong. If an exception is thrown by some user-provided operation, the `bimap` to which the set view belongs remains in its original state.

```
template< class KeyModifier >
bool modify_key(iterator position, KeyModifier mod);
```

- **Requires:** `KeyModifier` is a model of [Unary Function](#) accepting arguments of type: `key_type&`; `position` is a valid dereferenceable iterator of the view.

- **Effects:** Calls `mod(e.first)` where e is the element pointed to by position and rearranges `*position` into all the views of the `bimap`. If the rearrangement fails, the element is erased. Rearrangement is successful if

  - the map view is non-unique OR no other element with equivalent key exists,

  - AND rearrangement is allowed by all other views of the `bimap`.

- **Postconditions:** Validity of `position` is preserved if the operation succeeds.

- **Returns:** `true` if the operation succeeded, `false` otherwise.

- **Complexity:** O(M(n)).

- **Exception safety:** Basic. If an exception is thrown by some user-provided operation (except possibly mod), then the element pointed to by position is erased.

- **Note:** Only provided for map views.

```
template< class DataModifier >
bool modify_data(iterator position, DataModifier mod);
```

- **Requires:** `DataModifier` is a model of [Unary Function](#) accepting arguments of type: `mapped_type&`; `position` is a valid dereferenceable iterator of the view.

- **Effects:** Calls `mod(e.second)` where e is the element pointed to by position and rearranges `*position` into all the views of the `bimap`. If the rearrangement fails, the element is erased. Rearrangement is successful if

  - the oppositte map view is non-unique OR no other element with equivalent key in that view exists,

  - AND rearrangement is allowed by all other views of the `bimap`.

- **Postconditions:** Validity of `position` is preserved if the operation succeeds.

- **Returns:** `true` if the operation succeeded, `false` otherwise.

- **Complexity:** O(M(n)).

- **Exception safety:** Basic. If an exception is thrown by some user-provided operation (except possibly mod), then the element pointed to by position is erased.

- **Note:** Only provided for map views.

Lookup

`unordered_[multi]set_of` views provide the full lookup functionality required by unordered associative containers, namely `find`, `count`, and `equal_range`. Additionally, these member functions are templatized to allow for non-standard arguments, so extending the types of search operations allowed. The kind of arguments permissible when invoking the lookup member functions is defined by the following concept.

A type `CompatibleKey` is said to be a *compatible key* of (Hash, Pred) if (CompatibleKey, Hash, Pred) is a compatible extension of (Hash, Pred). This implies that `Hash` and `Pred` accept arguments of type `CompatibleKey`, which usually means they have several overloads of their corresponding `operator()` member functions.

```
template< class CompatibleKey >
iterator find(const CompatibleKey & x);

template< class CompatibleKey >
const_iterator find(const CompatibleKey & x) const;
```

- **Effects:** Returns a pointer to an element whose key is equivalent to x, or end() if such an element does not exist.

- **Complexity:** Average case O(1) (constant), worst case O(n).

```
template< class CompatibleKey >
size_type count(const CompatibleKey & x) const;
```

- **Effects:** Returns the number of elements with key equivalent to x.

- **Complexity:** Average case O(count(x)), worst case O(n).

```
template< class CompatibleKey >
std::pair<iterator,iterator>
    equal_range(const CompatibleKey & x);

template< class CompatibleKey >
std::pair<const_iterator,const_iterator>
    equal_range(const CompatibleKey & x) const;
```

- **Effects:** Returns a range containing all elements with keys equivalent to x (and only those).

- **Complexity:** Average case O(count(x)), worst case O(n).

at(), info_at() and operator[] - set_of only

```
template< class CompatibleKey >
const mapped_type & at(const CompatibleKey & k) const;
```

- **Requires:** CompatibleKey is a compatible key of key_compare.

- **Effects:** Returns the mapped_type reference that is associated with k, or throws std::out_of_range if such key does not exist.

- **Complexity:** Average case O(1) (constant), worst case O(n).

- **Note:** Only provided when unordered_set_of is used.

The symmetry of bimap imposes some constraints on operator[] and the non constant version of at() that are not found in std::maps. Tey are only provided if the other collection type is mutable (list_of, vector_of and unconstrained_set_of).

```
template< class CompatibleKey >
mapped_type & operator[](const CompatibleKey & k);
```

- **Requires:** CompatibleKey is a compatible key of key_compare.

- **Effects:** return insert(value_type(k,mapped_type()))->second;

- **Complexity:** If the insertion is performed O(I(n)), else: Average case O(1) (constant), worst case O(n).

- **Note:** Only provided when unordered_set_of is used and the other collection type is mutable.

```
template< class CompatibleKey >
mapped_type & at(const CompatibleKey & k);
```

- **Requires:** `CompatibleKey` is a compatible key of `key_compare`.

- **Effects:** Returns the `mapped_type` reference that is associated with `k`, or throws `std::out_of_range` if such key does not exist.

- **Complexity:** Average case O(1) (constant), worst case O(n).

- **Note:** Only provided when `unordered_set_of` is used and the other collection type is mutable.

```
template< class CompatibleKey >
info_type & info_at(const CompatibleKey & k);

template< class CompatibleKey >
const info_type & info_at(const CompatibleKey & k) const;
```

- **Requires:** `CompatibleKey` is a compatible key of `key_compare`.

- **Effects:** Returns the `info_type` reference that is associated with `k`, or throws `std::out_of_range` if such key does not exist.

- **Complexity:** Average case O(1) (constant), worst case O(n).

- **Note:** Only provided when `unordered_set_of` and `info_hook` are used

Hash policy

```
void rehash(size_type n);
```

- **Effects:** Increases if necessary the number of internal buckets so that `size()/bucket_count()` does not exceed the maximum load factor, and `bucket_count()>=n`.

- **Postconditions:** Validity of iterators and references to the elements contained is preserved.

- **Complexity:** Average case O(size()), worst case O(size(n)2).

- **Exception safety:** Strong.

Serialization

Views cannot be serialized on their own, but only as part of the `bimap` into which they are embedded. In describing the additional preconditions and guarantees associated to `unordered_[multi]set_of` views with respect to serialization of their embedding containers, we use the concepts defined in the `bimap` serialization section.

> **Operation:** saving of a `bimap` b to an output archive (XML archive) ar.

- **Requires:** No additional requirements to those imposed by the container.

> **Operation:** loading of a `bimap` b' from an input archive (XML archive) ar.

- **Requires:** Additionally to the general requirements, `key_eq()` must be serialization-compatible with `m.get<i>().key_eq()`, where i is the position of the `unordered_[multi]set_of` view in the container.

- **Postconditions:** On successful loading, the range `[begin(), end())` contains restored copies of every element in `[m.get<i>().begin(), m.get<i>().end())`, though not necessarily in the same order.

> **Operation:** saving of an `iterator` or `const_iterator` it to an output archive (XML archive) ar.

- **Requires:** `it` is a valid `iterator` of the view. The associated `bimap` has been previously saved.

> **Operation:** loading of an iterator or `const_iterator` it' from an input archive (XML archive) ar.

- **Postconditions:** On successful loading, if `it` was dereferenceable then `*it'` is the restored copy of `*it`, otherwise `it' == end()`.

- **Note:** It is allowed that `it` be a `const_iterator` and the restored `it'` an `iterator`, or viceversa.

> **Operation:** saving of a local_iterator or const_local_iterator it to an output archive (XML archive) ar.

- **Requires:** `it` is a valid local iterator of the view. The associated `bimap` has been previously saved.

> **Operation:** loading of a `local_iterator` or `const_local_iterator` it' from an input archive (XML archive) ar.

- **Postconditions:** On successful loading, if `it` was dereferenceable then `*it'` is the restored copy of `*it`; if `it` was `m.get<i>().end(n)` for some n, then `it' == m'.get<i>().end(n)` (where b is the original `bimap`, b' its restored copy and i is the ordinal of the index.)

- **Note:** It is allowed that `it` be a `const_local_iterator` and the restored `it'` a `local_iterator`, or viceversa.

list_of Reference

Header "boost/bimap/list_of.hpp" synopsis

```
namespace boost {
namespace bimaps {


template< class KeyType >
struct list_of;

struct list_of_relation;


} // namespace bimap
} // namespace boost
```

list_of Views

A list_of set view is a std::list signature compatible interface to the underlying heap of elements contained in a `bimap`.

If you look the bimap by a side, you will use a map view and if you looked it as a whole you will be using a set view.

Elements in a list_of view are by default sorted according to their order of insertion: this means that new elements inserted through a different view of the `bimap` are appended to the end of the list_of view. Additionally, the view allows for free reordering of elements in the same vein as `std::list` does. Validity of iterators and references to elements is preserved in all operations.

There are a number of differences with respect to `std::lists`:

- list_of views are not Assignable (like any other view.)

- Unlike as in `std::list`, insertions into a list_of view may fail due to clashings with other views. This alters the semantics of the operations provided with respect to their analogues in `std::list`.

- Elements in a list_of view are not mutable, and can only be changed by means of `replace` and `modify` member functions.

Having these restrictions into account, list_of views are models of Reversible Container, Front Insertion Sequence and Back Insertion Sequence. We only provide descriptions of those types and operations that are either not present in the concepts modeled or do not exactly conform to the requirements for these types of containers.

```cpp
namespace boost {
namespace bimaps {
namespace views {

template< -implementation defined parameter list- >
class -implementation defined view name-
{
    public:

    // types

    typedef -unspecified- value_type;
    typedef -unspecified- allocator_type;
    typedef -unspecified- reference;
    typedef -unspecified- const_reference;
    typedef -unspecified- iterator;
    typedef -unspecified- const_iterator;
    typedef -unspecified- size_type;
    typedef -unspecified- difference_type;
    typedef -unspecified- pointer;
    typedef -unspecified- const_pointer;
    typedef -unspecified- reverse_iterator;
    typedef -unspecified- const_reverse_iterator;

    typedef -unspecified- info_type;

    // construct/copy/destroy

    this_type & operator=(const this_type & x);

    template< class InputIterator >
    void assign(InputIterator first, InputIterator last);

    void assign(size_type n, const value_type & value);

    allocator_type get_allocator() const;

    // iterators

    iterator               begin();
    const_iterator         begin() const;

    iterator               end();
    const_iterator         end() const;

    reverse_iterator       rbegin();
    const_reverse_iterator rbegin() const;

    reverse_iterator       rend();
    const_reverse_iterator rend() const;

    // capacity

    bool     empty() const;

    size_type size() const;
```

```
size_type max_size() const;

void resize(size_type n, const value_type & x = value_type());

// access

const_reference front() const;
const_reference back() const;

// modifiers

std::pair<iterator,bool> push_front(const value_type & x);
void                     pop_front();

std::pair<iterator,bool> push_back(const value_type & x);
void                     pop_back();

std::pair<iterator,bool> insert(iterator position, const value_type & x);

void insert(iterator position, size_type n, const value_type & x);

template< class InputIterator >
void insert(iterator position, InputIterator first, InputIterator last);

iterator erase(iterator position);
iterator erase(iterator first, iterator last);

bool replace(iterator position, const value_type & x);

// Only in map views
// {

  typedef -unspecified- key_type;
  typedef -unspecified- mapped_type;
  typedef -unspecified- mapped_type; // Equal to mapped_type

  template< class CompatibleKey >
  bool replace_key(iterator position, const CompatibleKey & x);

  template< class CompatibleData >
  bool replace_data(iterator position, const CompatibleData & x);

  template< class KeyModifier >
  bool modify_key(iterator position, KeyModifier mod);

  template< class DataModifier >
  bool modify_data(iterator position, DataModifier mod);

// }


void clear();

// list operations

void splice(iterator position, this_type & x);
void splice(iterator position, this_type & x, iterator i);
void splice(
    iterator position, this_type & x, iterator first, iterator last);

void remove(const value_type & value);

template< class Predicate >
```

```
    void remove_if(Predicate pred);

    void unique();

    template< class BinaryPredicate >
    void unique(BinaryPredicate binary_pred);

    void merge(this_type & x);

    template< class Compare >
    void merge(this_type & x,Compare comp);

    void sort();

    template< class Compare >
    void sort(Compare comp);

    void reverse();

    // rearrange operations

    void relocate(iterator position, iterator i);
    void relocate(iterator position, iterator first, iterator last);

}

// view comparison

bool operator==(const this_type & v1, const this_type & v2 );
bool operator< (const this_type & v1, const this_type & v2 );
bool operator!=(const this_type & v1, const this_type & v2 );
bool operator> (const this_type & v1, const this_type & v2 );
bool operator>=(const this_type & v1, const this_type & v2 );
bool operator<=(const this_type & v1, const this_type & v2 );

} // namespace views
} // namespace bimap
} // namespace boost
```

In the case of a `bimap< list_of<Left>, ... >`

In the set view:

```
typedef signature-compatible with relation< Left, ... > key_type;
typedef signature-compatible with relation< Left, ... > value_type;
```

In the left map view:

```
typedef  Left  key_type;
typedef  ...   mapped_type;

typedef signature-compatible with std::pair< Left, ... > value_type;
```

In the right map view:

```
typedef  ...   key_type;
typedef  Left mapped_type;

typedef signature-compatible with std::pair< ... , Left > value_type;
```

Complexity signature

Here and in the descriptions of operations of `list_of` views, we adopt the scheme outlined in the complexity signature section. The complexity signature of a `list_of` view is:

- copying: `c(n) = n * log(n)`,

- insertion: `i(n) = 1` (constant),

- hinted insertion: `h(n) = 1` (constant),

- deletion: `d(n) = 1` (constant),

- replacement: `r(n) = 1` (constant),

- modifying: `m(n) = 1` (constant).

Instantiation types

`list_of` views are instantiated internally to `bimap` and specified by means of the collection type specifiers and the bimap itself. Instantiations are dependent on the following types:

- `Value` from `list_of`,

- `Allocator` from `bimap`,

Constructors, copy and assignment

As explained in the view concepts section, views do not have public constructors or destructors. Assignment, on the other hand, is provided.

```
this_type & operator=(const this_type & x);
```

- **Effects:** `a = b;` where a and b are the `bimap` objects to which `*this` and x belong, respectively.

- **Returns:** `*this`.

```
template< class InputIterator >
void assign(InputIterator first, InputIterator last);
```

- **Requires:** `InputIterator` is a model of Input Iterator over elements of type `value_type` or a type convertible to `value_type`. first and last are not iterators into any views of the `bimap` to which this view belongs. last is reachable from first.

- **Effects:** `clear(); insert(end(),first,last);`

```
void assign(size_type n, const value_type & value);
```

- **Effects:** `clear(); for(size_type i = 0; i < n ; ++n) push_back(v);`

Capacity operations

```
void resize(size_type n,const value_type& x=value_type());
```

- **Effects:** `if( n > size() ) insert(end(), n - size(), x); else if( n < size() ) {  iterator it = be-gin();  std::advance(it, n);  erase(it, end()); }`

- **Note:** If an expansion is requested, the size of the view is not guaranteed to be n after this operation (other views may ban insertions.)

Modifiers

```
std::pair<iterator,bool> push_front(const value_type& x);
```

- **Effects:** Inserts x at the beginning of the sequence if no other views of the bimap bans the insertion.

- **Returns:** The return value is a pair p. p.second is true if and only if insertion took place. On successful insertion, p.first points to the element inserted; otherwise, p.first points to an element that caused the insertion to be banned. Note that more than one element can be causing insertion not to be allowed.

- **Complexity:** O(I(n)).

- **Exception safety:** Strong.

```
std::pair<iterator,bool> push_back(const value_type & x);
```

- **Effects:** Inserts x at the end of the sequence if no other views of the bimap bans the insertion.

- **Returns:** The return value is a pair p. p.second is true if and only if insertion took place. On successful insertion, p.first points to the element inserted; otherwise, p.first points to an element that caused the insertion to be banned. Note that more than one element can be causing insertion not to be allowed.

- **Complexity:** O(I(n)).

- **Exception safety:** Strong.

```
std::pair<iterator,bool> insert(iterator position, const value_type & x);
```

- **Requires:** position is a valid iterator of the view.

- **Effects:** Inserts x before position if insertion is allowed by all other views of the bimap.

- **Returns:** The return value is a pair p. p.second is true if and only if insertion took place. On successful insertion, p.first points to the element inserted; otherwise, p.first points to an element that caused the insertion to be banned. Note that more than one element can be causing insertion not to be allowed.

- **Complexity:** O(I(n)).

- **Exception safety:** Strong.

```
void insert(iterator position, size_type n, const value_type & x);
```

- **Requires:** position is a valid iterator of the view.

- **Effects:** for(size_type i = 0; i < n; ++i) insert(position, x);

```
template< class InputIterator>
void insert(iterator position,InputIterator first,InputIterator last);
```

- **Requires:** position is a valid iterator of the view. InputIterator is a model of Input Iterator over elements of type value_type. first and last are not iterators into any view of the bimap to which this view belongs. last is reachable from first.

- **Effects:** while(first != last) insert(position, *first++);

- **Complexity:** O(m*I(n+m)), where m is the number of elements in [first,last).

- **Exception safety:** Basic.

```
iterator erase(iterator position);
```

- **Requires:** `position` is a valid dereferenceable `iterator` of the view.

- **Effects:** Deletes the element pointed to by `position`.

- **Returns:** An iterator pointing to the element immediately following the one that was deleted, or `end()` if no such element exists.

- **Complexity:** O(D(n)).

- **Exception safety:** nothrow.

```
iterator erase(iterator first, iterator last);
```

- **Requires:** `[first,last)` is a valid range of the view.

- **Effects:** Deletes the elements in `[first,last)`.

- **Returns:** `last`.

- **Complexity:** O(m*D(n)), where m is the number of elements in `[first,last)`.

- **Exception safety:** nothrow.

```
bool replace(iterator position,const value_type& x);
```

- **Requires:** `position` is a valid dereferenceable iterator of the view.

- **Effects:** Assigns the value x to the element pointed to by `position` into the `bimap` to which the view belongs if replacing is allowed by all other views of the `bimap`.

- **Postconditions:** Validity of `position` is preserved in all cases.

- **Returns:** `true` if the replacement took place, `false` otherwise.

- **Complexity:** O(R(n)).

- **Exception safety:** Strong. If an exception is thrown by some user-provided operation the `bimap` to which the view belongs remains in its original state.

```
template< class CompatibleKey >
bool replace_key(iterator position, const CompatibleKey & x);
```

- **Requires:** `position` is a valid dereferenceable iterator of the set view. `CompatibleKey` can be assigned to `key_type`.

- **Effects:** Assigns the value x to `e.first`, where e is the element pointed to by `position` into the `bimap` to which the set view belongs if replacing is allowed by all other views of the `bimap`.

- **Postconditions:** Validity of position is preserved in all cases.

- **Returns:** `true` if the replacement took place, `false` otherwise.

- **Complexity:** O(R(n)).

- **Exception safety:** Strong. If an exception is thrown by some user-provided operation, the `bimap` to which the set view belongs remains in its original state.

```
template< class CompatibleData >
bool replace_data(iterator position, const CompatibleData & x);
```

- **Requires:** `position` is a valid dereferenceable iterator of the set view. `CompatibleKey` can be assigned to `mapped_type`.

- **Effects:** Assigns the value `x` to `e.second`, where `e` is the element pointed to by `position` into the `bimap` to which the set view belongs if replacing is allowed by all other views of the `bimap`.

- **Postconditions:** Validity of position is preserved in all cases.

- **Returns:** `true` if the replacement took place, `false` otherwise.

- **Complexity:** O(R(n)).

- **Exception safety:** Strong. If an exception is thrown by some user-provided operation, the `bimap` to which the set view belongs remains in its original state.

```
template< class KeyModifier >
bool modify_key(iterator position, KeyModifier mod);
```

- **Requires:** `KeyModifier` is a model of Unary Function accepting arguments of type: `key_type&`; `position` is a valid dereferenceable iterator of the view.

- **Effects:** Calls `mod(e.first)` where `e` is the element pointed to by position and rearranges `*position` into all the views of the `bimap`. If the rearrangement fails, the element is erased. It is successful if the rearrangement is allowed by all other views of the `bimap`.

- **Postconditions:** Validity of `position` is preserved if the operation succeeds.

- **Returns:** `true` if the operation succeeded, `false` otherwise.

- **Complexity:** O(M(n)).

- **Exception safety:** Basic. If an exception is thrown by some user-provided operation (except possibly mod), then the element pointed to by position is erased.

- **Note:** Only provided for map views.

```
template< class DataModifier >
bool modify_data(iterator position, DataModifier mod);
```

- **Requires:** `DataModifier` is a model of Unary Function accepting arguments of type: `mapped_type&`; `position` is a valid dereferenceable iterator of the view.

- **Effects:** Calls `mod(e.second)` where `e` is the element pointed to by position and rearranges `*position` into all the views of the `bimap`. If the rearrangement fails, the element is erased. It is successful if the rearrangement is allowed by all other views of the `bimap`.

- **Postconditions:** Validity of `position` is preserved if the operation succeeds.

- **Returns:** `true` if the operation succeeded, `false` otherwise.

- **Complexity:** O(M(n)).

- **Exception safety:** Basic. If an exception is thrown by some user-provided operation (except possibly mod), then the element pointed to by position is erased.

- **Note:** Only provided for map views.

List operations

`list_of` views provide the full set of list operations found in `std::list`; the semantics of these member functions, however, differ from that of `std::list` in some cases as insertions might not succeed due to banning by other views. Similarly, the complexity of the operations may depend on the other views belonging to the same `bimap`.

```
void splice(iterator position, this_type & x);
```

- **Requires:** `position` is a valid iterator of the view. `&x!=this`.

- **Effects:** Inserts the contents of `x` before position, in the same order as they were in `x`. Those elements successfully inserted are erased from `x`.

- **Complexity:** O(`x.size()`*I(n+`x.size()`) + `x.size()`*D(`x.size()`)).

- **Exception safety:** Basic.

```
void splice(iterator position, this_type & x,iterator i);
```

- **Requires:** `position` is a valid iterator of the view. `i` is a valid dereferenceable iterator `x`.

- **Effects:** Inserts the element pointed to by `i` before position: if insertion is successful, the element is erased from `x`. In the special case `&x==this`, no copy or deletion is performed, and the operation is always successful. If `position==i`, no operation is performed.

- **Postconditions:** If `&x==this`, no iterator or reference is invalidated.

- **Complexity:** If `&x==this`, constant; otherwise O(I(n) + D(n)).

- **Exception safety:** If `&x==this`, nothrow; otherwise, strong.

```
void splice(iterator position, this_type & x, iterator first, iterator last);
```

- **Requires:** `position` is a valid iterator of the view. `first` and `last` are valid iterators of `x`. `last` is reachable from `first`. `position` is not in the range `[first,last)`.

- **Effects:** For each element in the range `[first,last)`, insertion is tried before position; if the operation is successful, the element is erased from x. In the special case `&x==this`, no copy or deletion is performed, and insertions are always successful.

- **Postconditions:** If `&x==this`, no iterator or reference is invalidated.

- **Complexity:** If `&x==this`, constant; otherwise O(m*I(n+m) + m*D(x.size())) where m is the number of elements in `[first,last)`.

- **Exception safety:** If `&x==this`, nothrow; otherwise, basic.

```
void remove(const value_type & value);
```

- **Effects:** Erases all elements of the view which compare equal to `value`.

- **Complexity:** O(n + m*D(n)), where m is the number of elements erased.

- **Exception safety:** Basic.

```
template< class Predicate >
void remove_if(Predicate pred);
```

- **Effects:** Erases all elements x of the view for which `pred(x)` holds.

93

- **Complexity:** O(n + m*D(n)), where m is the number of elements erased.

- **Exception safety:** Basic.

```
void unique();
```

- **Effects:** Eliminates all but the first element from every consecutive group of equal elements referred to by the iterator `i` in the range `[first+1,last)` for which `*i==*(i-1)`.

- **Complexity:** O(n + m*D(n)), where m is the number of elements erased.

- **Exception safety:** Basic.

```
template< class BinaryPredicate >
void unique(BinaryPredicate binary_pred);
```

- **Effects:** Eliminates all but the first element from every consecutive group of elements referred to by the iterator i in the range [first+1,last) for which `binary_pred(*i,*(i-1))` holds.

- **Complexity:** O(n + m*D(n)), where m is the number of elements erased.

- **Exception safety:** Basic.

```
void merge(this_type & x);
```

- **Requires:** `std::less<value_type>` is a Strict Weak Ordering over `value_type`. Both the view and x are sorted according to `std::less<value_type>`.

- **Effects:** Attempts to insert every element of x into the corresponding position of the view (according to the order). Elements successfully inserted are erased from x. The resulting sequence is stable, i.e. equivalent elements of either container preserve their relative position. In the special case `&x==this`, no operation is performed.

- **Postconditions:** Elements in the view and remaining elements in x are sorted. Validity of iterators to the view and of non-erased elements of x references is preserved.

- **Complexity:** If `&x==this`, constant; otherwise O(n + x.size()*I(n+x.size()) + x.size()*D(x.size())).

- **Exception safety:** If `&x==this`, nothrow; otherwise, basic.

```
template< class Compare >
void merge(this_type & x, Compare comp);
```

- **Requires:** Compare is a Strict Weak Ordering over `value_type`. Both the view and x are sorted according to `comp`.

- **Effects:** Attempts to insert every element of x into the corresponding position of the view (according to `comp`). Elements successfully inserted are erased from x. The resulting sequence is stable, i.e. equivalent elements of either container preserve their relative position. In the special case `&x==this`, no operation is performed.

- **Postconditions:** Elements in the view and remaining elements in x are sorted according to `comp`. Validity of iterators to the view and of non-erased elements of x references is preserved.

- **Complexity:** If `&x==this`, constant; otherwise O(n + x.size()*I(n+x.size()) + x.size()*D(x.size())).

- **Exception safety:** If `&x==this`, nothrow; otherwise, basic.

```
void sort();
```

- **Requires:** `std::less<value_type>` is a [Strict Weak Ordering](#) over value_type.

- **Effects:** Sorts the view according to `std::less<value_type>`. The sorting is stable, i.e. equivalent elements preserve their relative position.

- **Postconditions:** Validity of iterators and references is preserved.

- **Complexity:** O(n*log(n)).

- **Exception safety:** nothrow if `std::less<value_type>` does not throw; otherwise, basic.

```
template< typename Compare >
void sort(Compare comp);
```

- **Requires:** Compare is a [Strict Weak Ordering](#) over value_type.

- **Effects:** Sorts the view according to comp. The sorting is stable, i.e. equivalent elements preserve their relative position.

- **Postconditions:** Validity of iterators and references is preserved.

- **Complexity:** O(n*log(n)).

- **Exception safety:** nothrow if comp does not throw; otherwise, basic.

```
void reverse();
```

- **Effects:** Reverses the order of the elements in the view.

- **Postconditions:** Validity of iterators and references is preserved.

- **Complexity:** O(n).

- **Exception safety:** nothrow.

Rearrange operations

These operations, without counterpart in `std::list` (although splice provides partially overlapping functionality), perform individual and global repositioning of elements inside the index.

```
void relocate(iterator position, iterator i);
```

- **Requires:** `position` is a valid iterator of the view. `i` is a valid dereferenceable iterator of the view.

- **Effects:** Inserts the element pointed to by `i` before `position`. If `position==i`, no operation is performed.

- **Postconditions:** No iterator or reference is invalidated.

- **Complexity:** Constant.

- **Exception safety:** nothrow.

```
void relocate(iterator position, iterator first, iterator last);
```

- **Requires:** `position` is a valid iterator of the view. `first` and `last` are valid iterators of the view. `last` is reachable from `first`. `position` is not in the range `[first,last)`.

- **Effects:** The range of elements `[first,last)` is repositioned just before `position`.

- **Postconditions:** No iterator or reference is invalidated.

- **Complexity:** Constant.

- **Exception safety:** nothrow.

Serialization

Views cannot be serialized on their own, but only as part of the `bimap` into which they are embedded. In describing the additional preconditions and guarantees associated to `list_of` views with respect to serialization of their embedding containers, we use the concepts defined in the `bimap` serialization section.

> **Operation:** saving of a `bimap` b to an output archive (XML archive) ar.

- **Requires:** No additional requirements to those imposed by the container.

> **Operation:** loading of a `bimap` b' from an input archive (XML archive) ar.

- **Requires:** No additional requirements to those imposed by the container. **Postconditions:** On successful loading, each of the elements of `[begin(), end())` is a restored copy of the corresponding element in `[m.get<i>().begin(), m.get<i>().end())`, where `i` is the position of the `list_of` view in the container.

> **Operation:** saving of an `iterator` or `const_iterator` it to an output archive (XML archive) ar.

- **Requires:** `it` is a valid iterator of the view. The associated `bimap` has been previously saved.

> **Operation:** loading of an `iterator` or `const_iterator` it' from an input archive (XML archive) ar.

- **Postconditions:** On successful loading, if it was dereferenceable then `*it'` is the restored copy of `*it`, otherwise `it' == end()`.

- **Note:** It is allowed that `it` be a `const_iterator` and the restored `it'` an iterator, or viceversa.

vector_of Reference

Header "boost/bimap/vector_of.hpp" synopsis

```
namespace boost {
namespace bimaps {


template< class KeyType >
struct vector_of;

struct vector_of_relation;



} // namespace bimap
} // namespace boost
```

vector_of views

vector_of views are free-order sequences with constant time positional access and random access iterators. Elements in a vector_of view are by default sorted according to their order of insertion: this means that new elements inserted through a different view of the `bimap` are appended to the end of the vector_of view; additionally, facilities are provided for further rearrangement of the elements.

The public interface of vector_of views includes that of list_of views, with differences in the complexity of the operations, plus extra operations for positional access (`operator[]` and `at()`) and for capacity handling. Validity of iterators and references to elements is preserved in all operations, regardless of the capacity status.

As is the case with list_of views, vector_of views have the following limitations with respect to STL sequence containers:

- vector_of views are not Assignable (like any other view.)

- Insertions into a vector_of view may fail due to clashings with other views. This alters the semantics of the operations provided with respect to their analogues in STL sequence containers.

- Elements in a vector_of view are not mutable, and can only be changed by means of replace and modify member functions.

Having these restrictions into account, vector of views are models of Random Access Container and Back Insertion Sequence. Although these views do not model Front Insertion Sequence, because front insertion and deletion take linear time, front operations are nonetheless provided to match the interface of list_of views. We only describe those types and operations that are either not present in the concepts modeled or do not exactly conform to the requirements for these types of containers.

```cpp
namespace boost {
namespace bimaps {
namespace views {

template< -implementation defined parameter list- >
class -implementation defined view name-
{
    public:

    // types

    typedef -unspecified- value_type;
    typedef -unspecified- allocator_type;
    typedef -unspecified- reference;
    typedef -unspecified- const_reference;
    typedef -unspecified- iterator;
    typedef -unspecified- const_iterator;
    typedef -unspecified- size_type;
    typedef -unspecified- difference_type;
    typedef -unspecified- pointer;
    typedef -unspecified- const_pointer;
    typedef -unspecified- reverse_iterator;
    typedef -unspecified- const_reverse_iterator;

    typedef -unspecified- info_type;

    // construct / copy / destroy

    this_type & operator=(this_type & x);

    template< class InputIterator >
    void assign(InputIterator first, InputIterator last);

    void assign(size_type n, const value_type & value);

    allocator_type get_allocator() const;

    // iterators

    iterator                begin();
    const_iterator          begin() const;

    iterator                end();
    const_iterator          end() const;
```

```cpp
reverse_iterator        rbegin();
const_reverse_iterator rbegin() const;

reverse_iterator        rend();
const_reverse_iterator rend() const;

// capacity

bool      empty() const;

size_type size() const;

size_type max_size() const;

size_type capacity() const;

void reserve(size_type m);

void resize(size_type n, const value_type & x = value_type());

// access

const_reference operator[](size_type n) const;

const_reference at(size_type n) const;

const_reference front() const;

const_reference back() const;

// modifiers

std::pair<iterator,bool> push_front(const value_type & x);
void                     pop_front();

std::pair<iterator,bool> push_back(const value_type & x);
void                     pop_back();

std::pair<iterator,bool> insert(iterator position, const value_type & x);

void insert(iterator position, size_type m, const value_type & x);

template< class InputIterator>
void insert(iterator position, InputIterator first, InputIterator last);

iterator erase(iterator position);
iterator erase(iterator first, iterator last);

bool replace(iterator position, const value_type & x);

// Only in map views
// {
  typedef -unspecified- key_type;
  typedef -unspecified- mapped_type;
  typedef -unspecified- data_type; // Equal to mapped_type

  template< class CompatibleKey >
  bool replace_key(iterator position, const CompatibleKey & x);

  template< class CompatibleData >
  bool replace_data(iterator position, const CompatibleData & x);
```

```cpp
    template< class KeyModifier >
    bool modify_key(iterator position, KeyModifier mod);

    template< class DataModifier >
    bool modify_data(iterator position, DataModifier mod);

// }


void clear();

// list operations

void splice(iterator position, this_type & x);
void splice(iterator position, this_type & x, iterator i);
void splice(
    iterator position, this_type & x, iterator first, iterator last);

void remove(const value_type & value);

template< class Predicate >
void remove_if(Predicate pred);

void unique();

template< class BinaryPredicate >
void unique(BinaryPredicate binary_pred);

void merge(this_type & x);

template< typename Compare >
void merge(this_type & x, Compare comp);

void sort();

template< typename Compare >
void sort(Compare comp);

void reverse();

// rearrange operations

void relocate(iterator position, iterator i);
void relocate(iterator position, iterator first, iterator last);
};

// view comparison

bool operator==(const this_type & v1, const this_type & v2 );
bool operator< (const this_type & v1, const this_type & v2 );
bool operator!=(const this_type & v1, const this_type & v2 );
bool operator> (const this_type & v1, const this_type & v2 );
bool operator>=(const this_type & v1, const this_type & v2 );
bool operator<=(const this_type & v1, const this_type & v2 );

} // namespace views
} // namespace bimap
} // namespace boost
```

In the case of a `bimap< vector_of<Left>, ... >`

In the set view:

```
typedef signature-compatible with relation< Left, ... > key_type;
typedef signature-compatible with relation< Left, ... > value_type;
```

In the left map view:

```
typedef  Left  key_type;
typedef  ...    mapped_type;

typedef signature-compatible with std::pair< Left, ... > value_type;
```

In the right map view:

```
typedef  ...   key_type;
typedef  Left mapped_type;

typedef signature-compatible with std::pair< ... , Left > value_type;
```

Complexity signature

Here and in the descriptions of operations of `vector_of` views, we adopt the scheme outlined in the complexity signature section. The complexity signature of `vector_of` view is:

- copying: `c(n) = n * log(n)`,

- insertion: `i(n) = 1` (amortized constant),

- hinted insertion: `h(n) = 1` (amortized constant),

- deletion: `d(n) = m`, where m is the distance from the deleted element to the end of the sequence,

- replacement: `r(n) = 1` (constant),

- modifying: `m(n) = 1` (constant).

The following expressions are also used as a convenience for writing down some of the complexity formulas:

```
shl(a,b) = a+b if a is nonzero, 0 otherwise. rel(a,b,c) = if a<b, c-a, else a-b,
```

(`shl` and `rel` stand for *shift left* and *relocate*, respectively.)

Instantiation types

`vector_of` views are instantiated internally to `bimap` and specified by means of the collection type specifiers and the bimap itself. Instantiations are dependent on the following types:

- `Value` from `vector_of`,

- `Allocator` from `bimap`,

Constructors, copy and assignment

As explained in the views concepts section, views do not have public constructors or destructors. Assignment, on the other hand, is provided.

```
this_type & operator=(const this_type & x);
```

- **Effects:** `a=b;` where a and b are the `bimap` objects to which `*this` and x belong, respectively.

- **Returns:** `*this`.

```
template< class InputIterator >
void assign(InputIterator first, InputIterator last);
```

- **Requires:** `InputIterator` is a model of [Input Iterator](#) over elements of type `value_type` or a type convertible to `value_type`. `first` and `last` are not iterators into any view of the `bimap` to which this view belongs. `last` is reachable from `first`.

- **Effects:** `clear(); insert(end(),first,last);`

```
void assign(size_type n, const value_type & value);
```

- **Effects:** `clear(); for(size_type i = 0; i < n; ++n) push_back(v);`

Capacity operations

```
size_type capacity() const;
```

- **Returns:** The total number of elements c such that, when `size() < c`, back insertions happen in constant time (the general case as described by i(n) is *amortized* constant time.)

- **Note:** Validity of iterators and references to elements is preserved in all insertions, regardless of the capacity status.

```
void reserve(size_type m);
```

- **Effects:** If the previous value of `capacity()` was greater than or equal to `m`, nothing is done; otherwise, the internal capacity is changed so that `capacity()>=m`.

- **Complexity:** If the capacity is not changed, constant; otherwise O(n).

- **Exception safety:** If the capacity is not changed, nothrow; otherwise, strong.

```
void resize(size_type n, const value_type & x = value_type());
```

- **Effects:** `if( n > size() ) insert(end(), n-size(), x); else if( n<size() ) erase(begin()+n,end());`

- **Note:** If an expansion is requested, the size of the view is not guaranteed to be n after this operation (other views may ban insertions.)

Modifiers

```
std::pair<iterator,bool> push_front(const value_type & x);
```

- **Effects:** Inserts `x` at the beginning of the sequence if no other view of the `bimap` bans the insertion.

- **Returns:** The return value is a pair p. `p.second` is `true` if and only if insertion took place. On successful insertion, `p.first` points to the element inserted; otherwise, `p.first` points to an element that caused the insertion to be banned. Note that more than one element can be causing insertion not to be allowed.

- **Complexity:** O(n+I(n)).

- **Exception safety:** Strong.

```
std::pair<iterator,bool> push_back(const value_type & x);
```

- **Effects:** Inserts `x` at the end of the sequence if no other view of the `bimap` bans the insertion.

- **Returns:** The return value is a pair `p`. `p.second` is `true` if and only if insertion took place. On successful insertion, `p.first` points to the element inserted; otherwise, `p.first` points to an element that caused the insertion to be banned. Note that more than one element can be causing insertion not to be allowed.

- **Complexity:** O(I(n)).

- **Exception safety:** Strong.

```
std::pair<iterator,bool> insert(iterator position, const value_type & x);
```

- **Requires:** `position` is a valid iterator of the view.

- **Effects:** Inserts `x` before position if insertion is allowed by all other views of the `bimap`.

- **Returns:** The return value is a pair `p`. `p.second` is `true` if and only if insertion took place. On successful insertion, `p.first` points to the element inserted; otherwise, `p.first` points to an element that caused the insertion to be banned. Note that more than one element can be causing insertion not to be allowed.

- **Complexity:** O(shl(end()-position,1) + I(n)).

- **Exception safety:** Strong.

```
void insert(iterator position, size_type m, const value_type & x);
```

- **Requires:** `position` is a valid iterator of the view.

- **Effects:** `for(size_type i = 0; i < m; ++i) insert(position, x);`

- **Complexity:** O(shl(end()-position,m) + m*I(n+m)).

```
template< class InputIterator >
void insert(iterator position, InputIterator first, InputIterator last);
```

- **Requires:** `position` is a valid iterator of the view. `InputIterator` is a model of Input Iterator over elements of type `value_type` or a type convertible to `value_type`. `first` and `last` are not iterators into any view of the `bimap` to which this view belongs. `last` is reachable from `first`.

- **Effects:** `while(first!=last)insert(position,*first++);`

- **Complexity:** O(shl(end()-position,m) + m*I(n+m)), where m is the number of elements in `[first,last)`.

- **Exception safety:** Basic.

```
iterator erase(iterator position);
```

- **Requires:** `position` is a valid dereferenceable iterator of the view.

- **Effects:** Deletes the element pointed to by `position`.

- **Returns:** An iterator pointing to the element immediately following the one that was deleted, or `end()` if no such element exists.

- **Complexity:** O(D(n)).

- **Exception safety:** nothrow.

```
iterator erase(iterator first, iterator last);
```

- **Requires:** `[first,last)` is a valid range of the view.

- **Effects:** Deletes the elements in `[first,last)`.

- **Returns:** last.

- **Complexity:** O(m*D(n)), where m is the number of elements in `[first,last)`.

- **Exception safety:** nothrow.

```
bool replace(iterator position, const value_type & x);
```

- **Requires:** `position` is a valid dereferenceable iterator of the view.

- **Effects:** Assigns the value x to the element pointed to by position into the `bimap` to which the view belongs if replacing is allowed by all other views of the `bimap`.

- **Postconditions:** Validity of position is preserved in all cases.

- **Returns:** `true` if the replacement took place, `false` otherwise.

- **Complexity:** O(R(n)).

- **Exception safety:** Strong. If an exception is thrown by some user-provided operation the `bimap` to which the view belongs remains in its original state.

```
template< class CompatibleKey >
bool replace_key(iterator position, const CompatibleKey & x);
```

- **Requires:** `position` is a valid dereferenceable iterator of the set view. `CompatibleKey` can be assigned to `key_type`.

- **Effects:** Assigns the value x to `e.first`, where e is the element pointed to by `position` into the `bimap` to which the set view belongs if replacing is allowed by all other views of the `bimap`.

- **Postconditions:** Validity of position is preserved in all cases.

- **Returns:** `true` if the replacement took place, `false` otherwise.

- **Complexity:** O(R(n)).

- **Exception safety:** Strong. If an exception is thrown by some user-provided operation, the `bimap` to which the set view belongs remains in its original state.

```
template< class CompatibleData >
bool replace_data(iterator position, const CompatibleData & x);
```

- **Requires:** `position` is a valid dereferenceable iterator of the set view. `CompatibleKey` can be assigned to `mapped_type`.

- **Effects:** Assigns the value x to `e.second`, where e is the element pointed to by `position` into the `bimap` to which the set view belongs if replacing is allowed by all other views of the `bimap`.

- **Postconditions:** Validity of position is preserved in all cases.

- **Returns:** `true` if the replacement took place, `false` otherwise.

- **Complexity:** O(R(n)).

- **Exception safety:** Strong. If an exception is thrown by some user-provided operation, the `bimap` to which the set view belongs remains in its original state.

```
template< class KeyModifier >
bool modify_key(iterator position, KeyModifier mod);
```

- **Requires:** `KeyModifier` is a model of [Unary Function](#) accepting arguments of type: `key_type&`; `position` is a valid dereferenceable iterator of the view.

- **Effects:** Calls `mod(e.first)` where e is the element pointed to by position and rearranges `*position` into all the views of the `bimap`. If the rearrangement fails, the element is erased. It is successful if the rearrangement is allowed by all other views of the `bimap`.

- **Postconditions:** Validity of `position` is preserved if the operation succeeds.

- **Returns:** `true` if the operation succeeded, `false` otherwise.

- **Complexity:** O(M(n)).

- **Exception safety:** Basic. If an exception is thrown by some user-provided operation (except possibly mod), then the element pointed to by position is erased.

- **Note:** Only provided for map views.

```
template< class DataModifier >
bool modify_data(iterator position, DataModifier mod);
```

- **Requires:** `DataModifier` is a model of [Unary Function](#) accepting arguments of type: `mapped_type&`; `position` is a valid dereferenceable iterator of the view.

- **Effects:** Calls `mod(e.second)` where e is the element pointed to by position and rearranges `*position` into all the views of the `bimap`. If the rearrangement fails, the element is erased. It is successful if the rearrangement is allowed by all other views of the `bimap`.

- **Postconditions:** Validity of `position` is preserved if the operation succeeds.

- **Returns:** `true` if the operation succeeded, `false` otherwise.

- **Complexity:** O(M(n)).

- **Exception safety:** Basic. If an exception is thrown by some user-provided operation (except possibly mod), then the element pointed to by position is erased.

- **Note:** Only provided for map views.

List operations

`vector_of` views replicate the interface of `list_of` views, which in turn includes the list operations provided by `std::list`. The syntax and behavior of these operations exactly matches those of `list_of` views, but the associated complexity bounds differ in general.

```
void splice(iterator position, this_type & x);
```

- **Requires:** `position` is a valid iterator of the view. `&x!=this`.

- **Effects:** Inserts the contents of `x` before position, in the same order as they were in `x`. Those elements successfully inserted are erased from `x`.

- **Complexity:** O(shl(end()-position,x.size()) + x.size()*I(n+x.size()) + x.size()*D(x.size())).

- **Exception safety:** Basic.

```
void splice(iterator position, this_type & x,iterator i);
```

- **Requires:** `position` is a valid iterator of the view. `i` is a valid dereferenceable iterator `x`.

- **Effects:** Inserts the element pointed to by `i` before `position`: if insertion is successful, the element is erased from `x`. In the special case `&x==this`, no copy or deletion is performed, and the operation is always successful. If `position==i`, no operation is performed.

- **Postconditions:** If `&x==this`, no iterator or reference is invalidated.

- **Complexity:** If `&x==this`, O(rel(position,i,i+1)); otherwise O(shl(end()-position,1) + I(n) + D(n)).

- **Exception safety:** If `&x==this`, nothrow; otherwise, strong.

```
void splice(iterator position, this_type & x, iterator first, iterator last);
```

- **Requires:** `position` is a valid iterator of the view. `first` and `last` are valid iterators of `x`. `last` is reachable from `first`. `position` is not in the range `[first,last)`.

- **Effects:** For each element in the range `[first,last)`, insertion is tried before `position`; if the operation is successful, the element is erased from `x`. In the special case `&x==this`, no copy or deletion is performed, and insertions are always successful.

- **Postconditions:** If `&x==this`, no iterator or reference is invalidated.

- **Complexity:** If `&x==this`, O(rel(position,first,last)); otherwise O(shl(end()-position,m) + m*I(n+m) + m*D(x.size())) where m is the number of elements in `[first,last)`.

- **Exception safety:** If `&x==this`, nothrow; otherwise, basic.

```
void remove(const value_type & value);
```

- **Effects:** Erases all elements of the view which compare equal to `value`.

- **Complexity:** O(n + m*D(n)), where m is the number of elements erased.

- **Exception safety:** Basic.

```
template< class Predicate >
void remove_if(Predicate pred);
```

- **Effects:** Erases all elements `x` of the view for which `pred(x)` holds.

- **Complexity:** O(n + m*D(n)), where m is the number of elements erased.

- **Exception safety:** Basic.

```
void unique();
```

- **Effects:** Eliminates all but the first element from every consecutive group of equal elements referred to by the iterator `i` in the range `[first+1,last)` for which `*i==*(i-1)`.

- **Complexity:** O(n + m*D(n)), where m is the number of elements erased.

- **Exception safety:** Basic.

```
template< class BinaryPredicate >
void unique(BinaryPredicate binary_pred);
```

- **Effects:** Eliminates all but the first element from every consecutive group of elements referred to by the iterator i in the range `[first+1,last)` for which `binary_pred(*i, *(i-1))` holds.

- **Complexity:** O(n + m*D(n)), where m is the number of elements erased.

- **Exception safety:** Basic.

```
void merge(this_type & x);
```

- **Requires:** `std::less<value_type>` is a Strict Weak Ordering over `value_type`. Both the view and x are sorted according to `std::less<value_type>`.

- **Effects:** Attempts to insert every element of x into the corresponding position of the view (according to the order). Elements successfully inserted are erased from x. The resulting sequence is stable, i.e. equivalent elements of either container preserve their relative position. In the special case `&x==this`, no operation is performed.

- **Postconditions:** Elements in the view and remaining elements in x are sorted. Validity of iterators to the view and of non-erased elements of x references is preserved.

- **Complexity:** If `&x==this`, constant; otherwise O(n + x.size()*I(n+x.size()) + x.size()*D(x.size())).

- **Exception safety:** If `&x==this`, nothrow; otherwise, basic.

```
template< class Compare >
void merge(this_type & x, Compare comp);
```

- **Requires:** `Compare` is a Strict Weak Ordering over `value_type`. Both the view and x are sorted according to comp.

- **Effects:** Attempts to insert every element of x into the corresponding position of the view (according to `comp`). Elements successfully inserted are erased from x. The resulting sequence is stable, i.e. equivalent elements of either container preserve their relative position. In the special case `&x==this`, no operation is performed.

- **Postconditions:** Elements in the view and remaining elements in x are sorted according to `comp`. Validity of iterators to the view and of non-erased elements of x references is preserved.

- **Complexity:** If `&x==this`, constant; otherwise O(n + x.size()*I(n+x.size()) + x.size()*D(x.size())).

- **Exception safety:** If `&x==this`, nothrow; otherwise, basic.

```
void sort();
```

- **Requires:** `std::less<value_type>` is a Strict Weak Ordering over `value_type`.

- **Effects:** Sorts the view according to `std::less<value_type>`. The sorting is stable, i.e. equivalent elements preserve their relative position.

- **Postconditions:** Validity of iterators and references is preserved.

- **Complexity:** O(n*log(n)).

- **Exception safety:** Basic.

```
template< class Compare >
void sort(Compare comp);
```

- **Requires:** Compare is a [Strict Weak Ordering](#) over `value_type`.

- **Effects:** Sorts the view according to `comp`. The sorting is stable, i.e. equivalent elements preserve their relative position.

- **Postconditions:** Validity of iterators and references is preserved.

- **Complexity:** O(n*log(n)).

- **Exception safety:** Basic.

```
void reverse();
```

- **Effects:** Reverses the order of the elements in the view.

- **Postconditions:** Validity of iterators and references is preserved.

- **Complexity:** O(n).

- **Exception safety:** nothrow.

Rearrange operations

These operations, without counterpart in `std::list` (although splice provides partially overlapping functionality), perform individual and global repositioning of elements inside the index.

```
void relocate(iterator position, iterator i);
```

- **Requires:** `position` is a valid iterator of the view. `i` is a valid dereferenceable iterator of the view.

- **Effects:** Inserts the element pointed to by `i` before `position`. If `position==i`, no operation is performed.

- **Postconditions:** No iterator or reference is invalidated.

- **Complexity:** Constant.

- **Exception safety:** nothrow.

```
void relocate(iterator position, iterator first, iterator last);
```

- **Requires:** `position` is a valid iterator of the view. `first` and `last` are valid iterators of the view. `last` is reachable from `first`. `position` is not in the range `[first,last)`.

- **Effects:** The range of elements `[first,last)` is repositioned just before `position`.

- **Postconditions:** No iterator or reference is invalidated.

- **Complexity:** Constant.

- **Exception safety:** nothrow.

Serialization

Views cannot be serialized on their own, but only as part of the `bimap` into which they are embedded. In describing the additional preconditions and guarantees associated to `vector_of` views with respect to serialization of their embedding containers, we use the concepts defined in the `bimap` serialization section.

> **Operation:** saving of a `bimap` b to an output archive (XML archive) ar.

- **Requires:** No additional requirements to those imposed by the container.

> **Operation:** loading of a `bimap` b' from an input archive (XML archive) ar.

- **Requires:** No additional requirements to those imposed by the container.

- **Postconditions:** On successful loading, each of the elements of `[begin(), end())` is a restored copy of the corresponding element in `[m.get<i>().begin(), m.get<i>().end())`, where `i` is the position of the `vector_of` view in the container.

> **Operation:** saving of an `iterator` or `const_iterator` `it` to an output archive (XML archive) ar.

- **Requires:** `it` is a valid iterator of the view. The associated `bimap` has been previously saved.

> **Operation:** loading of an `iterator` or `const_iterator` `it'` from an input archive (XML archive) ar.

- **Postconditions:** On successful loading, if it was dereferenceable then `*it'` is the restored copy of `*it`, otherwise `it'==end()`.

- **Note:** It is allowed that it be a `const_iterator` and the restored `it'` an `iterator`, or viceversa.

unconstrained_set_of Reference

Header "boost/bimap/unconstrained_set_of.hpp" synopsis

```
namespace boost {
namespace bimaps {


template< class KeyType >
struct unconstrained_set_of;

struct unconstrained_set_of_relation;


} // namespace bimap
} // namespace boost
```

unconstrained_set_of Views

An unconstrained_set_of set view is a view with no constraints. The use of these kind of view boost the bimap performance but the view can not be accessed. An unconstrained view is an empty class.

```
namespace boost {
namespace bimaps {
namespace views {

template< -implementation defined parameter list- >
class -implementation defined view name-
{
    // Empty view
};

} // namespace views
} // namespace bimap
} // namespace boost
```

In the case of a `bimap< unconstrained_set_of<Left>, ... >`

In the set view:

```
typedef signature-compatible with relation< Left, ... > key_type;
typedef signature-compatible with relation< Left, ... > value_type;
```

In the left map view:

```
typedef  Left  key_type;
typedef  ...   mapped_type;

typedef signature-compatible with std::pair< Left, ... > value_type;
```

In the right map view:

```
typedef  ...   key_type;
typedef  Left mapped_type;

typedef signature-compatible with std::pair< ... , Left > value_type;
```

Complexity signature

We adopt the scheme outlined in the complexity signature section. An unconstrained view can not be accessed by the user, but the formulas to find the order of an operation for a bimap hold with the following definitions. The complexity signature of a `uncon-strained_set_of` view is:

- copying: `c(n) = 0`

- insertion: `i(n) = 0`

- hinted insertion: `h(n) = 0`

- deletion: `d(n) = 0`

- replacement: `r(n) = 0`

- modifying: `m(n) = 0`

Serialization

Views cannot be serialized on their own, but only as part of the `bimap` into which they are embedded. In describing the additional preconditions and guarantees associated to `list_of` views with respect to serialization of their embedding containers, we use the concepts defined in the `bimap` serialization section.

> **Operation:** saving of a `bimap` b to an output archive (XML archive) ar.

- **Requires:** No additional requirements to those imposed by the container.

> **Operation:** loading of a `bimap` b' from an input archive (XML archive) ar.

- **Requires:** No additional requirements to those imposed by the container.

Compiler specifics

Compiler	OS Tested	State
GCC 3.3	Linux	Supported
GCC 3.4	Linux	Supported
GCC 4.0	Linux, Mac	Supported
GCC 4.1	Linux	Supported
GCC 4.2	Linux	Supported
ICC 8.0	Linux	Supported
ICC 9.0	Linux	Supported
ICC 9.1	Linux	Supported
GCC 4.2	Linux	Supported
GCC 4.2	Linux	Supported
VS 7.1	Windows	Supported
VS 8.0	Windows	Supported
ICC 7.1	Windows	Not Supported
ICC 8.0	Windows	Supported
ICC 9.1	Windows	Supported
CW 8.3	Windows	Not Supported

VS 7.1

If a .cpp file uses more than four differents bimaps the compiler will run out of symbols and issue an internal compiler error. The official solution in msdn is to split the .cpp in several files or upgrade your compiler.

VS 8.0

VC++ 8.0 warns on usage of certain Standard Library and API functions that can be cause buffer overruns or other possible security issues if misused. See http://msdn.microsoft.com/msdnmag/issues/05/05/SafeCandC/default.aspx But the wording of the warning is misleading and unsettling, there are no portable alternative functions, and VC++ 8.0's own libraries use the functions in question. In order to turn off the warnings add the followings defines at the begging of your .cpp files:

```
#define _CRT_SECURE_NO_DEPRECATE
#define _SCL_SECURE_NO_DEPRECATE
```

Performance

Section under construction.

Examples

Examples list

In the folder libs/bimap/example you can find all the examples used in bimap documentation. Here is a list of them:

Table 7. Tutorial examples

Program	Description
simple_bimap.cpp	Soccer world cup example
tagged_simple_bimap.cpp	Soccer world cup example using user defined names
step_by_step.cpp	Basic example of the three views of bimap
population_bimap.cpp	Countries populations, using `unordered_set_of` and `multiset_of`
repetitions_counter.cpp	Word repetitions counter, using `unordered_set_of` and `list_of`
mighty_bimap.cpp	Dictionary using `list_of_relation`
user_defined_names.cpp	Equivalence between code with tagged and untagged code
standard_map_comparison.cpp	Comparison between standard maps and bimap map views
at_function_examples.cpp	Functions `at(key)` and `operator[](key)` examples
tutorial_modify_and_replace.cpp	`modify` and `replace` examples
tutorial_range.cpp	`range()` tutorial
tutorial_info_hook.cpp	Additional information hooking
unconstrained_collection.cpp	Using `unconstrained_set_of` collection type

Table 8. Bimap and Boost examples

Program	Description
assign.cpp	Bimap and Boost.Assign: Methods to insert elements
lambda.cpp	Bimap and Boost.Lambda: new lambda placeholders
property_map.cpp	Bimap and Boost.PropertyMap: PropertyMap support
range.cpp	Bimap and Boost.Range: Using bimaps in the new range framework
foreach.cpp	Bimap and Boost.Foreach: Iterating over bimaps
typeof.cpp	Bimap and Boost.Typeof: using BOOST_AUTO while we wait for C++0x
xpressive.cpp	Bimap and Boost.Xpressive: Inserting elements in a bimap
serialization.cpp:	Bimap and Boost.Serialization: Load and save bimaps and iterators

Table 9. Boost.MultiIndex to Boost.Bimap path examples

Program	Description
bidirectional_map.cpp	Boost.MultiIndex to Boost.Bimap path example
hashed_indices.cpp	Boost.MultiIndex to Boost.Bimap path example
tagged_bidirectional_map.cpp	Boost.MultiIndex to Boost.Bimap path example

Simple Bimap

This is the example from the one minute tutorial section.

Go to source code

```cpp
#include <string>
#include <iostream>

#include <boost/bimap.hpp>

template< class MapType >
void print_map(const MapType & map,
               const std::string & separator,
               std::ostream & os )
{
    typedef typename MapType::const_iterator const_iterator;

    for( const_iterator i = map.begin(), iend = map.end(); i != iend; ++i )
    {
        os << i->first << separator << i->second << std::endl;
    }
}

int main()
{
    // Soccer World cup

    typedef boost::bimap< std::string, int > results_bimap;
    typedef results_bimap::value_type position;

    results_bimap results;
    results.insert( position("Argentina"    ,1) );
    results.insert( position("Spain"        ,2) );
    results.insert( position("Germany"      ,3) );
    results.insert( position("France"       ,4) );

    std::cout << "The number of countries is " << results.size()
              << std::endl;

    std::cout << "The winner is " << results.right.at(1)
              << std::endl
              << std::endl;

    std::cout << "Countries names ordered by their final position:"
              << std::endl;

    // results.right works like a std::map< int, std::string >

    print_map( results.right, ") ", std::cout );

    std::cout << std::endl
              << "Countries names ordered alphabetically along with"
                 "their final position:"
              << std::endl;

    // results.left works like a std::map< std::string, int >

    print_map( results.left, " ends in position ", std::cout );

    return 0;
}
```

You can rewrite it using tags to gain readability.

Go to source code

```cpp
#include <iostream>

#include <boost/bimap.hpp>

struct country  {};
struct place    {};

int main()
{
    using namespace boost::bimaps;

    // Soccer World cup.

    typedef bimap
    <
        tagged< std::string, country >,
        tagged< int         , place   >

    > results_bimap;

    typedef results_bimap::value_type position;

    results_bimap results;
    results.insert( position("Argentina"    ,1) );
    results.insert( position("Spain"        ,2) );
    results.insert( position("Germany"      ,3) );
    results.insert( position("France"       ,4) );

    std::cout << "Countries names ordered by their final position:"
              << std::endl;

  ❶for( results_bimap::map_by<place>::const_iterator
          i    = results.by<place>().begin(),
          iend = results.by<place>().end() ;
          i != iend; ++i )
    {
      ❷std::cout << i->get<place  >() << ") "
                << i->get<country>() << std::endl;
    }

    std::cout << std::endl
              << "Countries names ordered alfabetically along with"
                 "their final position:"
              << std::endl;

  ❸for( results_bimap::map_by<country>::const_iterator
          i    = results.by<country>().begin(),
          iend = results.by<country>().end() ;
          i != iend; ++i )
    {
        std::cout << i->get<country>() << " ends "
                  << i->get<place  >() << "°"
                  << std::endl;
    }

    return 0;
}
```

❶   `results.by<place>()` is equivalent to `results.right`
❷   `get<Tag>` works for each view of the bimap
❸   `results.by<country>()` is equivalent to `results.left`

Mighty Bimap

This is the translator example from the tutorial. In this example the collection type of relation is changed to allow the iteration of the container.

Go to source code

```cpp
#include <iostream>
#include <string>
#include <boost/bimap/bimap.hpp>
#include <boost/bimap/list_of.hpp>
#include <boost/bimap/unordered_set_of.hpp>

struct english {};
struct spanish {};

int main()
{
    using namespace boost::bimaps;

    typedef bimap
    <
        unordered_set_of< tagged< std::string, spanish > >,
        unordered_set_of< tagged< std::string, english > >,
        list_of_relation

    > translator;

    translator trans;

    // We have to use `push_back` because the collection of relations is
    // a `list_of_relation`

    trans.push_back( translator::value_type("hola"  ,"hello"   ) );
    trans.push_back( translator::value_type("adios" ,"goodbye" ) );
    trans.push_back( translator::value_type("rosa"  ,"rose"    ) );
    trans.push_back( translator::value_type("mesa"  ,"table"   ) );

    std::cout << "enter a word" << std::endl;
    std::string word;
    std::getline(std::cin,word);

    // Search the queried word on the from index (Spanish)

    translator::map_by<spanish>::const_iterator is
        = trans.by<spanish>().find(word);

    if( is != trans.by<spanish>().end() )
    {
        std::cout << word << " is said "
                  << is->get<english>()
                  << " in English" << std::endl;
    }
    else
    {
        // Word not found in Spanish, try our luck in English

        translator::map_by<english>::const_iterator ie
            = trans.by<english>().find(word);

        if( ie != trans.by<english>().end() )
        {
```

116

```
        std::cout << word << " is said "
                  << ie->get<spanish>()
                  << " in Spanish" << std::endl;
    }
    else
    {
        // Word not found, show the possible translations

        std::cout << "No such word in the dictionary"      << std::endl;
        std::cout << "These are the possible translations" << std::endl;

        for( translator::const_iterator
                i = trans.begin(),
                i_end = trans.end();

                i != i_end ; ++i )
        {
            std::cout << i->get<spanish>()
                      << " <---> "
                      << i->get<english>()
                      << std::endl;
        }
    }
}
    return 0;
}
```

MultiIndex to Bimap Path - Bidirectional Map

This is example 4 in Boost.MultiIndex documentation.

This example shows how to construct a bidirectional map with multi_index_container. By a bidirectional map we mean a container of elements of `std::pair<const FromType,const ToType>` such that no two elements exists with the same first or second value (`std::map` only guarantees uniqueness of the first member). Fast look-up is provided for both keys. The program features a tiny Spanish-English dictionary with on-line query of words in both languages.

Boost.MultiIndex

Go to source code

```cpp
#include <iostream>
#include <boost/tokenizer.hpp>

#include <boost/multi_index_container.hpp>
#include <boost/multi_index/key_extractors.hpp>
#include <boost/multi_index/ordered_index.hpp>

using namespace boost;
using namespace boost::multi_index;

// tags for accessing both sides of a bidirectional map

struct from {};
struct to   {};

// The class template bidirectional_map wraps the specification
// of a bidirectional map based on multi_index_container.

template<typename FromType,typename ToType>
struct bidirectional_map
{
    typedef std::pair<FromType,ToType> value_type;

    typedef multi_index_container<
        value_type,
        indexed_by
        <
            ordered_unique
            <
                tag<from>, member<value_type,FromType,&value_type::first>
            >,
            ordered_unique
            <
                tag<to>, member<value_type,ToType,&value_type::second>
            >
        >

  > type;

};

// A dictionary is a bidirectional map from strings to strings

typedef bidirectional_map<std::string,std::string>::type dictionary;

int main()
{
    dictionary d;

    // Fill up our microdictionary.
    // first members Spanish, second members English.

    d.insert(dictionary::value_type("hola","hello"));
    d.insert(dictionary::value_type("adios","goodbye"));
    d.insert(dictionary::value_type("rosa","rose"));
    d.insert(dictionary::value_type("mesa","table"));

    std::cout << "enter a word" << std::endl;
    std::string word;
    std::getline(std::cin,word);

    // search the queried word on the from index (Spanish)
```

```
    dictionary::iterator it = d.get<from>().find(word);

    if( it != d.end() )
    {
        // the second part of the element is the equivalent in English

        std::cout << word << " is said "
                  << it->second << " in English" << std::endl;
    }
    else
    {
        // word not found in Spanish, try our luck in English

        dictionary::index_iterator<to>::type it2 = d.get<to>().find(word);
        if( it2 != d.get<to>().end() )
        {
            std::cout << word << " is said "
                      << it2->first << " in Spanish" << std::endl;
        }
        else
        {
            std::cout << "No such word in the dictionary" << std::endl;
        }
    }

    return 0;
}
```

Boost.Bimap

Go to source code

```cpp
#include <iostream>
#include <boost/tokenizer.hpp>
#include <boost/bimap/bimap.hpp>

using namespace boost::bimaps;

// A dictionary is a bidirectional map from strings to strings

typedef bimap<std::string,std::string> dictionary;
typedef dictionary::value_type translation;

int main()
{
    dictionary d;

    // Fill up our microdictionary.
    // first members Spanish, second members English.

    d.insert( translation("hola" ,"hello"  ));
    d.insert( translation("adios","goodbye"));
    d.insert( translation("rosa" ,"rose"   ));
    d.insert( translation("mesa" ,"table"  ));

    std::cout << "enter a word" << std::endl;
    std::string word;
    std::getline(std::cin,word);

    // search the queried word on the from index (Spanish)

    dictionary::left_const_iterator it = d.left.find(word);

    if( it != d.left.end() )
    {
        // the second part of the element is the equivalent in English

        std::cout << word << " is said "
                  << it->second ❶
                  << " in English" << std::endl;
    }
    else
    {
        // word not found in Spanish, try our luck in English

        dictionary::right_const_iterator it2 = d.right.find(word);
        if( it2 != d.right.end() )
        {
            std::cout << word << " is said "
                      << it2->second ❷
                      << " in Spanish" << std::endl;
        }
        else
        {
            std::cout << "No such word in the dictionary" << std::endl;
        }
    }

    return 0;
}
```

❶   `it` is an iterator of the left view, so `it->second` refers to the right element of the relation, the word in english
❷   `it2` is an iterator of the right view, so `it2->second` refers to the left element of the relation, the word in spanish

Or better, using tags...

Go to source code

```cpp
#include <iostream>

#include <boost/bimap/bimap.hpp>

using namespace boost::bimaps;

// tags

struct spanish {};
struct english {};

// A dictionary is a bidirectional map from strings to strings

typedef bimap
<
    tagged< std::string,spanish >, tagged< std::string,english >

> dictionary;

typedef dictionary::value_type translation;

int main()
{
    dictionary d;

    // Fill up our microdictionary.
    // first members Spanish, second members English.

    d.insert( translation("hola" ,"hello"  ));
    d.insert( translation("adios","goodbye"));
    d.insert( translation("rosa" ,"rose"   ));
    d.insert( translation("mesa" ,"table"  ));

    std::cout << "enter a word" << std::endl;
    std::string word;
    std::getline(std::cin,word);

    // search the queried word on the from index (Spanish) */

    dictionary::map_by<spanish>::const_iterator it =
        d.by<spanish>().find(word);

    if( it != d.by<spanish>().end() )
    {
        std::cout << word << " is said "
                  << it->get<english>() << " in English" << std::endl;
    }
    else
    {
        // word not found in Spanish, try our luck in English

        dictionary::map_by<english>::const_iterator it2 =
            d.by<english>().find(word);

        if( it2 != d.by<english>().end() )
        {
            std::cout << word << " is said "
                      << it2->get<spanish>() << " in Spanish" << std::endl;
        }
```

```
        else
        {
            std::cout << "No such word in the dictionary" << std::endl;
        }
    }

    return 0;
}
```

MultiIndex to Bimap Path - Hashed indices

This is example 8 of Boost.MultiIndex.

> Hashed indices can be used as an alternative to ordered indices when fast look-up is needed and sorting information is of no interest. The example features a word counter where duplicate entries are checked by means of a hashed index.

Boost.MultiIndex

Go to source code

```cpp
#include <iostream>
#include <iomanip>

#include <boost/tokenizer.hpp>

#include <boost/multi_index_container.hpp>
#include <boost/multi_index/key_extractors.hpp>
#include <boost/multi_index/ordered_index.hpp>
#include <boost/multi_index/hashed_index.hpp>
#include <boost/lambda/lambda.hpp>

using namespace boost::multi_index;
namespace bl = boost::lambda;

// word_counter keeps the ocurrences of words inserted. A hashed
// index allows for fast checking of preexisting entries.

struct word_counter_entry
{
    std::string  word;
    unsigned int occurrences;

    word_counter_entry( std::string word_ ) : word(word_), occurrences(0) {}
};

typedef multi_index_container
<
    word_counter_entry,
    indexed_by
    <
        ordered_non_unique
        <
            BOOST_MULTI_INDEX_MEMBER(
                word_counter_entry,unsigned int,occurrences),
            std::greater<unsigned int>
        >,
        hashed_unique
        <
            BOOST_MULTI_INDEX_MEMBER(word_counter_entry,std::string,word)
        >
    >

> word_counter;

typedef boost::tokenizer<boost::char_separator<char> > text_tokenizer;

int main()
{
    std::string text=
        "En un lugar de la Mancha, de cuyo nombre no quiero acordarme... "
        "...snip..."
        "...no se salga un punto de la verdad.";

    // feed the text into the container

    word_counter   wc;
    text_tokenizer tok(text,boost::char_separator<char>(" \t\n.,;:!?'\"-"));
    unsigned int   total_occurrences = 0;

    for( text_tokenizer::iterator it = tok.begin(), it_end = tok.end();
         it != it_end ; ++it )
    {
        ++total_occurrences;
```

```
        word_counter::iterator wit = wc.insert(*it).first;
        wc.modify_key( wit, ++ bl::_1 );
    }

    // list words by frequency of appearance

    std::cout << std::fixed << std::setprecision(2);

    for( word_counter::iterator wit = wc.begin(), wit_end=wc.end();
         wit != wit_end; ++wit )
    {
        std::cout << std::setw(11) << wit->word << ": "
                  << std::setw(5)
                  << 100.0 * wit->occurrences / total_occurrences << "%"
                  << std::endl;
    }

    return 0;
}
```

Boost.Bimap

Go to source code

```cpp
#include <iostream>
#include <iomanip>

#include <boost/tokenizer.hpp>

#include <boost/bimap/bimap.hpp>
#include <boost/bimap/unordered_set_of.hpp>
#include <boost/bimap/multiset_of.hpp>
#include <boost/bimap/support/lambda.hpp>

using namespace boost::bimaps;

struct word        {};
struct occurrences {};

typedef bimap
<

    multiset_of< tagged<unsigned int,occurrences>, std::greater<unsigned int> >,
unordered_set_of< tagged< std::string,       word>                            >

> word_counter;

typedef boost::tokenizer<boost::char_separator<char> > text_tokenizer;

int main()
{

    std::string text=
        "Relations between data in the STL are represented with maps."
        "A map is a directed relation, by using it you are representing "
        "a mapping. In this directed relation, the first type is related to "
        "the second type but it is not true that the inverse relationship "
        "holds. This is useful in a lot of situations, but there are some "
        "relationships that are bidirectional by nature.";

    // feed the text into the container

    word_counter   wc;
    text_tokenizer tok(text,boost::char_separator<char>(" \t\n.,;:!?'\"-"));
    unsigned int   total_occurrences = 0;

    for( text_tokenizer::const_iterator it = tok.begin(), it_end = tok.end();
         it != it_end ; ++it )
    {
        ++total_occurrences;

        word_counter::map_by<occurrences>::iterator wit =
            wc.by<occurrences>().insert(
                word_counter::map_by<occurrences>::value_type(0,*it)
            ).first;

        wc.by<occurrences>().modify_key( wit, ++_key);
    }

    // list words by frequency of appearance

    std::cout << std::fixed << std::setprecision(2);

    for( word_counter::map_by<occurrences>::const_iterator
            wit     = wc.by<occurrences>().begin(),
            wit_end = wc.by<occurrences>().end();
```

```
       wit != wit_end; ++wit )
   {
       std::cout << std::setw(15) << wit->get<word>() << ": "
                 << std::setw(5)
                 << 100.0 * wit->get<occurrences>() / total_occurrences << "%"
                 << std::endl;
   }

   return 0;
}
```

Test suite

The Boost.Bimap test suite exercises the whole spectrum of functionalities provided by the library. Although the tests are not meant to serve as a learning guide, the interested reader may find it useful to inspect the source code to gain familiarity with some of the least common features offered by Boost.Bimap.

Program	Description
test_tagged.cpp	Tagged idiom checks
test_mutant.cpp	Test the mutant idiom
test_structured_pair.cpp	Test structured pair class
test_mutant_relation.cpp	Test the relation class
test_bimap_set_of.cpp	Library interface check
test_bimap_multiset_of.cpp	Library interface check
test_bimap_unordered_set_of.cpp	Library interface check
test_bimap_unordered_multiset_of.cpp	Library interface check
test_bimap_list_of.cpp	Library interface check
test_bimap_vector_of.cpp	Library interface check
test_bimap_convenience_header.cpp	Library interface check
test_bimap_ordered.cpp	Test set and multiset based bimaps
test_bimap_unordered.cpp	Test unordered_set and unordered_multiset based bimaps
test_bimap_sequenced.cpp	Test list and vector based bimaps
test_bimap_unconstrained.cpp	Test bimaps with unconstrained views
test_bimap_serialization.cpp	Serialization support checks
test_bimap_property_map.cpp	Property map concepts for the set and unordered set views
test_bimap_modify.cpp	`replace`, `modify` and `operator[]`
test_bimap_lambda.cpp	Test lambda modified idom support
test_bimap_assign.cpp	Test Boost.Assign support
test_bimap_project.cpp	Projection of iterators support
test_bimap_operator_bracket.cpp	`operator[]` and `at()` functions
test_bimap_info.cpp	Information hooking support
test_bimap_extra.cpp	Additional checks
test_bimap_info_1.cpp	Information hooking compilation fail test
test_bimap_info_2.cpp	Information hooking compilation fail test
test_bimap_info_3.cpp	Information hooking compilation fail test
test_bimap_mutable_1.cpp	Mutable members compilation fail test

Program	Description
test_bimap_mutable_2.cpp	Mutable members compilation fail test
test_bimap_mutable_3.cpp	Mutable members compilation fail test

Future work

Rearrange Function

Boost.MultiIndex includes some others functions that can be included in the views.

Release notes

Boost 1.53 release

- Refactor map_view_iterator to improve error messages.

- Fixed replace_(left/right/key/data) bug when using relations with info.

- Fixed vector_of and list_of views `assign(size_type,value_type)` bug.

- Fixed vector_of views `operator[](size_type)` and `at(size_type)` functions return value.

Boost 1.52 release

- Fixed unused parameter name warnings in Boost.Bimap (#7425).

Boost 1.50 release

- Added key_type and mapped_type (leaving data_type for backward compatibility) typedefs to map views (#6031).

- Removed spurious dependency on Boost.Serialization (#3868).

- Solved unused constructor parameter warning (#5749).

Boost 1.45 release

- Fixed custom allocators (#3665).

- Fixed count() constness (#2484).

- Other maintenance fixes.

Boost 1.35 release

- Added to Boost Libraries.

Rationale

This section assumes that you have read all other sections, the most of important of which being *tutorial*, *std::set theory* and the *reference*, and that you have tested the library. A lot of effort was invested in making the interface as intuitive and clean as possible. If you understand, and hopefully like, the interface of this library, it will be a lot easier to read this rationale. The following section is little more than a rationale. This library was coded in the context of the Google SoC 2006 and the student and mentor were in different continents. A great deal of email flowed between Joaquin and Matias. The juiciest parts of the conversations where extracted and rearranged here.

> **Note**
>
> To browse the code, you can use the *Bimap Complete Reference*, a doxygen-powered document targeted at developers.

General Design

The initial explanation includes few features. This section aims to describe the general design of the library and excludes details of those features that are of lesser importance; these features will be introduced later.

The design of the library is divided into two parts. The first is the construction of a `relation` class. This will be the object stored and managed by the `multi_index_container` core. The idea is to make this class as easy as possible to use, while making it efficient in terms of memory and access time. This is a cornerstone in the design of **Boost.Bimap** and, as you will see in this rationale, the rest of the design follows easily.

The following interface is necessary for the `relation` class:

```cpp
typedef -unspecified- TA; typedef -unspecified- TB;

TA a, ai; TB b, bi;

typedef relation< TA, TB > rel;
STATIC_ASSERT( is_same< rel::left_type , TA >::value );
STATIC_ASSERT( is_same< rel::right_type, TB >::value );

rel r(ai,bi);
assert( r.left == ai && r.right == bi );

r.left  = a; r.right = b;
assert( r.left  == a && r.right == b );

typedef pair_type_by< member_at::left , rel >::type pba_type;
STATIC_ASSERT( is_same< pba_type::first_type , TA >::value );
STATIC_ASSERT( is_same< pba_type::second_type, TB >::value );

typedef pair_type_by< member_at::right, rel >::type pbb_type;
STATIC_ASSERT( is_same< pbb_type::first_type , TB >::value );
STATIC_ASSERT( is_same< pbb_type::second_type, TA >::value );

pba_type pba = pair_by< member_at::left  >(r);
assert( pba.first == r.left  && pba.second == r.right );

pbb_type pbb = pair_by< member_at::right >(r);
assert( pbb.first == r.right && pbb.second == r.left  );
```
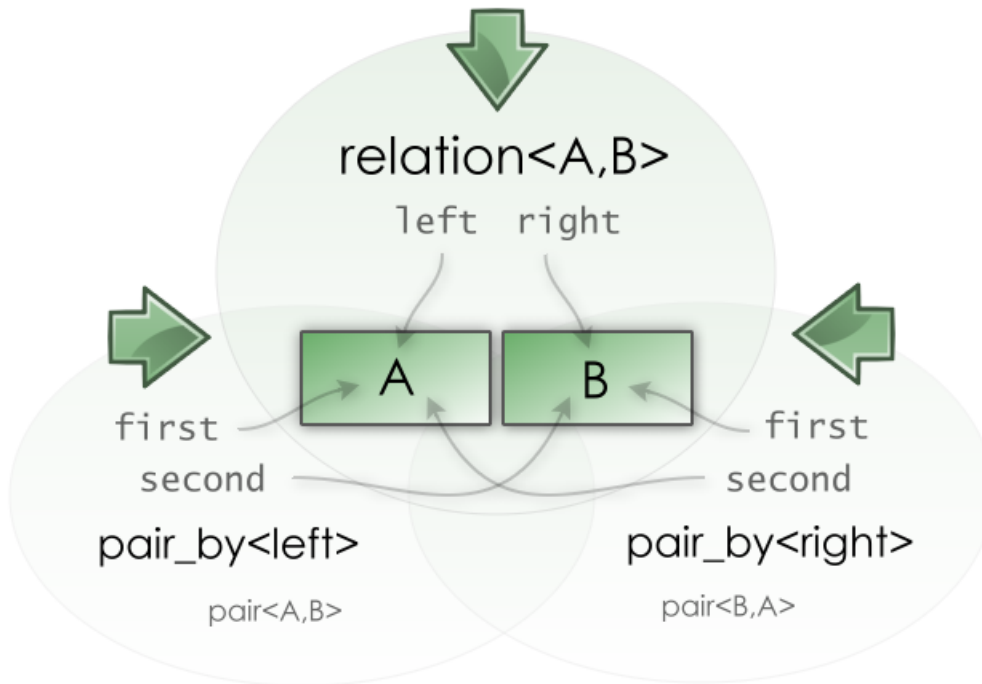
Although this seems straightforward, as will be seen later, it is the most difficult code hack of the library. It is indeed very easy if we relax some of the efficiency constraints. For example, it is trivial if we allow a relation to have greater size than the the sum of those of its components. It is equally simple if access speed is not important. One of the first decisions made about **Boost.Bimap** was, however, that, in order to be useful, it had to achieve zero overhead over the wrapped **Boost.MultiIndex** container. Finally, there is another constraint that can be relaxed: conformance to C++ standards, but this is quite unacceptable. Let us now suppose that we have coded this class, and it conforms to what was required.

The second part is based on this `relation` class. We can now view the data in any of three ways: `pair<A,B>`, `relation<A,B>` and `pair<B,A>`. Suppose that our bimap supports only one-to-one relations. (Other relation types are considered additional features in this design.) The proposed interface is very simple, and it is based heavily on the concepts of the STL. Given a `bimap<A,B> bm`:

1. `bm.left` is signature-compatible with a `std::map<A,B>`

2. `bm.right` is signature-compatible with a `std::map<B,A>`

3. `bm` is signature-compatible with a `std::set<relation<A,B> >`
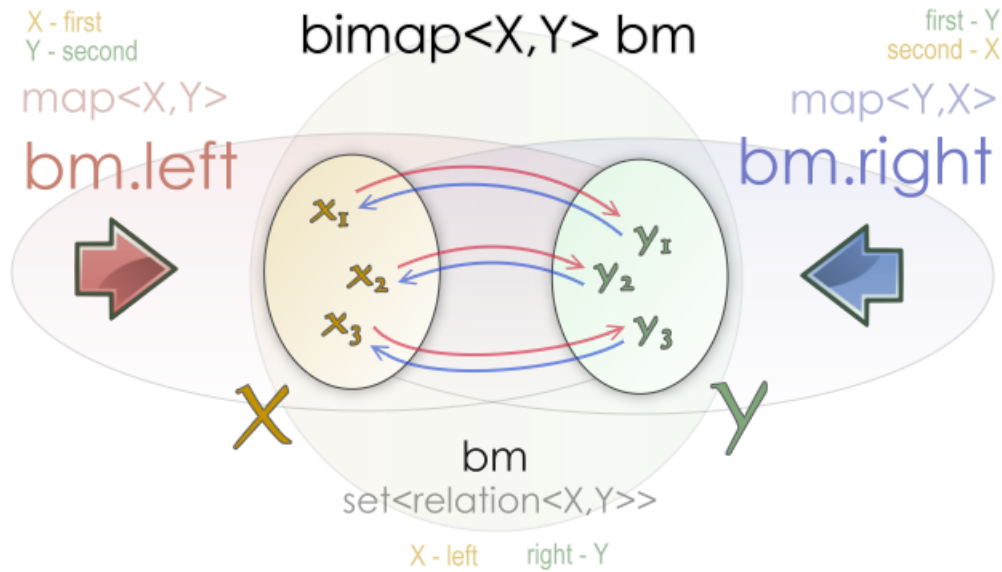
This interface is easily learned by users who have a STL background, as well as being simple and powerful. This is the general design.

Relation Implementation

This section explains the details of the actual `relation` class implementation.

The first thing that we can imagine is the use of an `union`. Regrettably, the current C++ standard only allows unions of POD types. For the views, we can try a wrapper around a `relation<A,B>` that has two references named first and second that bind to `A` and `B`, or to `B` and `A`.

```
relation<TA,TB> r;

const_reference_pair<A,B> pba(r);
const_reference_pair<B,A> pbb(r);
```

It is not difficult to code the relation class using this, but two references are initialized at every access and using of `pba.first` will be slower in most compilers than using `r.left` directly . There is another hidden drawback of using this scheme: it is not iterator-friendly, since the map views iterators must be degraded to *Read Write* instead of *LValue*. This will be explained later.

At first, this seems to be the best we can do with the current C++ standard. However there is a solution to this problem that does not conform very well to C++ standards but does achieve zero overhead in terms of access time and memory, and additionally allows the view iterators to be upgraded to *LValue* again.

In order to use this, the compiler must conform to a layout-compatibility clause that is not currently in the standard but is very natural. The additional clause imposes that if we have two classes:

```
struct class_a_b
{
    Type1 name_a;
    Type2 name_b;
};

struct class_b_a
{
    Type1 name_b;
    Type2 name_a;
};
```

then the storage layout of `class_a_b` is equal to the storage layout of `class_b_a`. If you are surprised to learn that this does not hold in a standards-compliant C++ compiler, welcome to the club. It is the natural way to implement it from the point of view of the compiler's vendor and is very useful for the developer. Maybe it will be included in the standard some day. Every current compiler conforms to this.

If we are able to count on this, then we can implement an idiom called `mutant`. The idea is to provide a secure wrapper around `reinterpret_cast`. A class can declare that it can be viewed using different view classes that are storage-compatible with it. Then we use the free function `mutate<view>(mutant)` to get the view. The `mutate` function checks at compile time that the requested view is declared in the mutant views list. We implement a class name `structured_pair` that is signature-compatible with a `std::pair`, while the storage layout is configured with a third template parameter. Two instances of this template class will provide the views of the relation.

The thing is that if we want to be standards-compliant, we cannot use this approach. It is very annoying not to be able to use something that we know will work with every compiler and that is far better than alternatives. So -- and this is an important decision -- we have to find a way to use it and still make the library standards-compliant.

The idea is very simple. We code both approaches: the const_reference_pair-based and the mutant-based, and use the mutant approach if the compiler is compliant with our new layout-compatible clause. If the compiler really messes things up, we degrade the performance of the bimap a little. The only drawback here is that, while the mutant approach allows to make *LValue* iterators, we have to degrade them to *Read Write* in both cases, because we require that the same code be compilable by any standards-compliant compiler.

> **Note**
>
> Testing this approach in all the supported compilers indicated that the mutant idiom was always supported. The strictly compliant version was removed from the code because it was never used.

Bimap Implementation

The core of bimap will be obviously a `multi_index_container`. The basic idea to tackle the implementation of the bimap class is to use `iterator_adaptor` to convert the iterators from Boost.MultiIndex to the `std::map` and `std::set` behaviour. The `map_view` and the `set_view` can be implemented directly using this new transformed iterators and a wrapper around each index of the core container. However, there is a hidden idiom here, that, once coded, will be very useful for other parts of this library and for Boost.MRU library. Following the ideas from `iterator_adaptor`, Boost.Bimap views are implemented using a `container_adaptor`. There are several template classes (for example `map_adaptor` and `set_adaptor`) that take a `std::map` signature-conformant class and new iterators, and adapt the container so it now uses this iterators instead of the originals. For example, if you have a `std::set<int*>`, you can build other container that behaves exactly as a `std::set<int>` using `set_adaptor` and `iterator_adaptor`. The combined use of this two tools is very powerful. A `container_adaptor` can take classes that do not fulfil all the requirements of the adapted container. The new container must define these missing functions.

Additional Features

N-1, N-N, hashed maps

This is a very interesting point of the design. The framework introduced in *std::set theory* permits the management of the different constraints with a very simple and conceptual approach. It is easy both to remember and to learn. The idea here is to allow the user to specify the collection type of each key directly. In order to implement this feature, we have to solve two problems:

• The index types of the `multi_index_container` core now depends on the collection type used for each key.

• The map views now change their semantics according to the collection type chosen.

Boost.Bimap relies heavily on Boost.MPL to implement all of the metaprogramming necessary to make this framework work. By default, if the user does not specify the kind of the set, a `std::set` type is used.
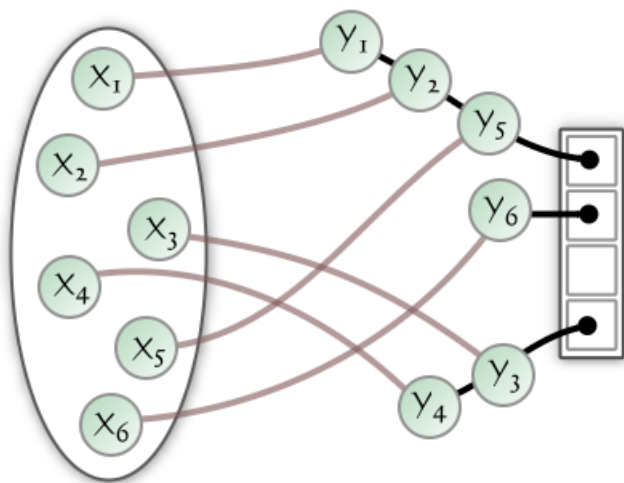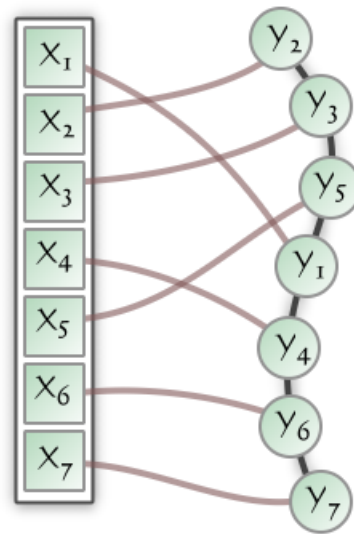
bimap<set_of<X>,unordered_set_of<Y>>   bimap<vector_of<X>,list_of<Y>>

Collection type of relation constraints

The constraints of the bimap set view are another very important feature. In general, Boost.Bimap users will base the set view type on one of the two collection types of their keys. It may be useful however to give this set other constraints or simply to order it differently. By default, Boost.Bimap bases the collection type of relations on the left collection type, but the user may choose between:

- left_based

- right_based

- set_of_relation<>

- multiset_of_relation<>

- unordered_set_of_relation<>

- unordered_multiset_of_relation<>

- list_of

- vector_of

In the first two cases, there are only two indices in the `multi_index_core`, and for this reason, these are the preferred options. The implementation uses further metaprogramming to define a new index if necessary.

Tagged

The idea of using tags instead of the `member_at::side` idiom is very appealing since code that uses it is more readable. The only cost is compile time. *boost/bimap/tagged* is the implementation of a non-invasive tagged idiom. The `relation` class is built in such a way that even when the user uses tags, the `member_at::side` idiom continues to work. This is good since an user can start tagging even before completing the coding of the algorithm, and the untagged code continues to work. The development becomes a little more complicated when user-defined tags are included, but there are many handy metafunctions defined in the `tagged` idiom that help to keep things simple enough.
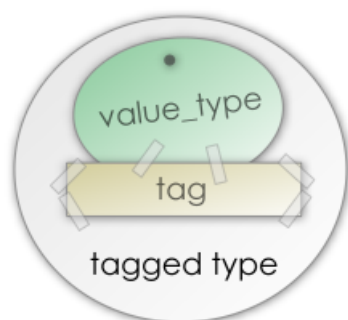
Code

You can browse the code using the **Boost.Bimap doxygen docs**.

The code follows the Boost Library Requirement and Guidelines as closely as possible.

Table 10. folders in boost/bimap

name	what is inside?
user level header files	
tagged/	tagged idiom
relation/	the bimap data
container_adaptor/	easy way of adapting containers
views/	bimap views
property_map/	support for property map concept

Table 11. folders in each folder

name	what is inside?
	class definitions
support/	optional metafunctions and free functions
detail/	things not intended for the user's eyes

The student and the mentor

> **Tip**
>
> It is a good idea to read the original Boost.Misc SoC proposal first.

```
- The discussion starts with Joaquin trying to strip out the "misc" name out of the
library -
```

Joaquin

Thinking about it, the unifying principle of MISC containers is perhaps misleading: certainly all miscs use multi-indexing internally, but this does not reflect much in the external interface (as it should be, OTOH). So, from the user's point of view, miscs are entirely heterogeneous beasts. Moreover, there isn't in your proposal any kind of global facility common to all miscs. What about dropping the misc principle and working on each container as a separate library, then? You'd have boost::bimap, boost::mru, etc, and no common intro to them. This also opens up the possibility to add other containers to the suite which aren't based on B.MI. What's your stance on this? Do you see a value in keeping miscs conceptually together?

Matias

As the original proposal states only two containers (bimap and mru set) both based in B.MI, it was straight forward to group them together. When I was writing the SoC proposal I experienced a similar feeling when the two families begin to grow. As you say, the only common denominator is their internal implementation. I thought a bit about a more general framework to join this two families (and other internally related ones) and finally came up with an idea: Boost.MultiIndex! So I think that it is not a good idea to try to unify the two families and I voted in favor of get rid of the misc part of boost::misc::bimap and boost::misc::mru. Anyway, for my SoC application it seems OK to put the two families in the same project because although from the outside they are completely unrelated, the work I will have to do in order to build the libraries will be consistent and what I will learn coding the bimap family will be used when I start to code the mru family. When the mru family is in place, I will surely have learnt other things to improve the bimap group.

On the other hand, I think it will be useful for the general user to have at least some document linked in the B.MI documentation that enumerates the most common cases of uses (a bimap and an mru set for example) and points where to find clean implementation for this useful containers. For now, a link to boost::bimap and other one to boost::mru will suffice. If you think about the title of such a document, you will probably come up with something like: Common Multi Index Specialized Containers, and we are back to our misc proposal. So, to order some ideas:

- A new family of containers that can be accessed by both key will be created. (boost::bimap)

- A new family of time aware containers will see the light. (boost::mru)

- A page can be added to B.MI documentation, titled misc that links this new libraries.

This is a clearer framework for the user. They can use a mru container without hearing about Boost.MultiIndex at all. And B.MI users will get some of their common containers already implemented with an STL friendly interface in other libraries. And as you stated this is more extensible because opens the door to use other libraries in bimap and mru families than just Boost.MultiIndex without compromising the more general boost framework. The word "misc" it is going to disappear from the code and the documentation of bimap and mru. From now on the only use for it will be to identify our SoC project. I am thinking in a name for the bimap library. What about Boost.BidirectionalMap? Ideas?

Joaquin

Yes, Boost.Bimap. In my opinion, bimap is a well known name in the Boost and even in the C++ community. It sounds and is short. Why not to vindicate yourself as the owner of this name?

```
- Then after a week of work -
```

Matias

> Now that Boost.Bimap is getting some shape, I see that as you have told me, we must offer a "one_to_many_map" and a "multi_bimap" as part of the library. The framework I am actually working allowed to construct this kind of bidirectional maps and it is easy to understand from the user side.

Joaquin

> OK, I am glad we agree on this point.

Matias

> With respect to the symmetry of the key access names, I have to agree that there is not much a difference between the following ones:

> - to - from

> - to - b

> - 0 - 1

> - left - right

> In my opinion it is a matter of taste, but left/right sounds more symmetrical than the others.

Joaquin

> I like very much the left/right notation, it is very simple to remember and it is a lot more symmetrical than to/from.

Matias

> At first my idea was to obtain ease of use hiding the B.MI core, making it more STL-intuitive. Nevertheless I have realized that B.MI is a lot more coherent and easy to use that I had imagined. This makes me think again in the problem. In the design that I am coding now, bimap **is-a** multi_index_container specializes with a data type very comfortable called bipair, that can be seen like any of the two maps that integrates it using map views. This scheme has great benefits for users:

> - If the user already knows B.MI, he can take advantage of the tools that it provides and that are not present in the STL containers. In addition, in some cases the use to indices to see the data can be very useful.

> - If the user does not know anything about B.MI but have an STL framework, the learning curve is reduced to understand the bimap instantiation and how a is obtained the desired map view.

> Another very important benefit holds: All the algorithms done for B.MI continues to work with Boost.Bimap and if B.MI continues growing, bimap grow automatically.

Joaquin

> Umm... This is an interesting design decision, but controversial in my opinion. Basically you decide to expose the implementation of bimap; that has advantages, as you stated, but also a nonsmall disadvantage: once **you have documented** the implementation, it is not possible to change it anymore. It is a marriage with B.MI without the chance of divorce. The other possibility, to hide the implementation and to duplicate and document the provided functionality, explicitly or implicitly due to the same characteristics of the implementation, is of course heavier to maintain, but it gives a degree of freedom to change the guts of your software if you need to. Do not take this like a frontal objection, but I think that it is quite important design decision, not only in the context of bimap but in general.

Matias

You are quite right here. I think we have to choose the hardest path and hide the B.MI core from the user. I am sending you the first draft of bimap along with some documentation.

```
- This completes the second week, the documentation was basically the first section of this rationale
-
```

Joaquin

I must confess that I am beginning to like what I see. I am mathematical by vocation, and when I see symmetry in a formulation I believe that it is in the right track.

Matias

We are two mathematicians by vocation then.

Joaquin

I think that the part of std::set theory is very clear. To me, it turns out to me somewhat strange to consider the rank of a map (values X) like a std::set, but of course the formulation is consistent.

Matias

I like it very much, it can be a little odd at first, but now that I have get used to it, it is very easy to express in the code my contrains on the data, and I believe that if somebody reads the code and sees the bimap instantiation he is not going to have problems understanding it. Perhaps it is easier to understand it if we use your notation: ordered_nonunique, unordered_unique, but this goes against our STL facade. In my opinion the user that comes from STL must have to learn as less as possible.

Joaquin

Considering a relation like a `struct {left, right}` *is clean and clear. If I understand it well, one relation has views of type* `pair{first, second}`*, is this correct?*

Matias

Yes, I believe that the left/right notation to express symmetry is great. I believe that to people is going to love it.

Joaquin

OK, perfect. I likes this very much:

- bm.left is compatible with std::map<A,B>

- bm.right is compatible with std::map<B,A>

- bm is compatible with std::set<relation<A,B>>

It is elegant and symmetric. I feel good vibrations here.

Matias

Great!

Joaquin

Moving on, the support for N-1, N-N, and hashed index is very easy to grasp, and it fits well in framework. However I do not finish to understand very well the "set<relation> constraints" section. Will you came up with some examples of which is the meaning of the different cases that you enumerate?

Matias -

Yes, I mean:

- based on the left

- based on the right

The bimap core must be based on some index of multi index. If the index of the left is of the type hash, then in fact the main view is going to be an unordered_set< relation<A,B> >. Perhaps this is not what the user prefers and he wants to base its main view on the right index.

- set_of_relation

- multiset_of_relation

- unordered_set_of_relation

- unordered_multiset_of_relation

However, if both of them are hash indexes, the user may want the main view to be ordered. As we have a B.MI core this is very easy to support, we just have to add another index to it.

Joaquin

I understand it now. OK, I do not know if we have to include this in the first version, is going to be a functionality avalanche!

Matias

The user is not affected by the addition of this functionality, because by default it will be based on the left index that is a very natural behaviour. I do not think that this is functionality bloat, but I agree with you that it is a functionality avalanche.

Joaquin

There are restrictions between the left and right set types and the possible main view set types. For example if some of the index is of unique type, then the main view cannot be of type multiset_of_relation. To the inverse one, if the main view is of type set_of_relation the left and the right index cannot be of type multi_set. All this subject of the unicity constrictions and the resulting interactions between indexes is one of the subtle subjects of B.MI.

Matias

This can be checked at compile time and informed as an error in compile time.

Joaquin

It can be interesting.

- And right when everything seems to be perfect... -

Joaquin

I have some worse news with respect to mutant, it is very a well designed and manageable class, unfortunately, C++ does not guarantee layout-compatibility almost in any case. For example, the C++ standard does not guarantee that the classes `struct{T1 a; T2 b;}` *and* `struct{T1 b; T2 a;}` *are layout-compatible, and therefore the trick of reinterpret_cast is an undefined behavior. I am with you in which that in the 100% of the cases this scheme will really work, but the standard is the standard. If you can look the layout-compatibility subject in it (http://www.kuzbass.ru/docs/isocpp/). As you see, sometimes the standard is cruel. Although mutant seems a lost case, please do not hurry to eliminate it. We will see what can be done for it.*

Matias

I read the standard, and you were right about it. Mutant was an implementation detail. It is a pity because I am sure that it will work perfect in any compiler. Perhaps the standard becomes more strict some day and mutant

returns to life... We can then try a wrapper around a relation<A,B> that have two references named first and second that bind to A and B, or B and A.

```
relation<TA,TB> r;
const_reference_pair<A,B> pba(r);
const_reference_pair<B,A> pbb(r);
```

It is not difficult to code the relation class in this way but two references are initialized with every access and the use of `pba.first` *will be slower than* `r.left` *in most compilers. It is very difficult to optimize this kind of references.*

Joaquin

This workaround is not possible, due to technical problems with the expected behavior of the iterators. If the iterators of bm.left are of bidirectional type, then standard stated that it have to return an object of type const value_type& when dereferenced. You will have to return a const_reference_pair created in the flight, making it impossible to return a reference.

Matias

I understand... I have workaround for that also but surely the standard will attack me again! We must manage to create the class relation that responds as we want, the rest of the code will flow from this point. This clear separation between the relation class and the rest of the library, is going to help to us to separate the problems and to attack them better.

Joaquin

What workaround? It already pricks my curiosity,I have dedicated a long time to the subject and I do not find any solution except that we allow the relation class to occupy more memory.

Matias

We must achieve that the relation<A,B> size equals the pair<A,B> size if we want this library to be really useful. I was going to write my workaround and I realized that It does not work. Look at this: http://www.boost.org/libs/iterator/doc/new-iter-concepts.html Basically the problem that we are dealing is solved if we based our iterators on this proposal. The present standard forces that the bidirectional iterators also are of the type input and output. Using the new concepts there is no inconvenient in making our iterators "Readable Writable Swappable Bidirectional Traversal". Therefore the const_reference_pair returns to be valid.

Joaquin

It is correct in the sense that you simply say that your iterators are less powerful than those of the std::map. It is not that it is wrong, simply that instead of fixing the problem, you confess it.

Matias

OK, but in our particular case; What are the benefits of offering a LValue iterator against a Read Write iterator? It does not seem to me that it is less powerful in this case.

Joaquin

The main problem with a ReadWrite is that the following thing: `value_type * p=&(*it);` *fails or stores a transitory direction in p. Is this important in the real life? I do not know. How frequently you store the direction of the elements of a map? Perhaps it is not very frequent, since the logical thing is to store the iterators instead of the directions of the elements. Let us review our options:*

1. We used mutant knowing that is not standard, but of course it is supported in the 100% of the cases.

2. We used const_reference_pair and we declared the iterators not LValue.

---

3. We found some trick that still we do not know. I have thus been playing with unions and things, without much luck.

4. We leverage the restriction that views have to support the first, second notation. If we made this decision, there are several possibilities:

a. The left map has standard semantics first/second while the right map has the inverse semantics.

b. Instead of first and second we provide first() and second(), with which the problem is trivial.

c. The map view do not support first/second but left/right as the father relation

5. We solve the problem using more memory than sizeof(pair<A,B>).

In any case, I would say that the only really unacceptable option is the last one.

Matias

Lets see.

1. I want the "standard compliant" label in the library.

2. This is the natural choice, but knowing that there is another option that always works and it is more efficient is awful.

3. I have also tried to play with unions, the problem is that the union members must be POD types.

4. This option implies a big lost to the library.

5. Totally agree.

I want to add another option to this list. Using metaprogramming, the relation class checks if the compiler supports the mutant idiom. If it supports it then it uses it and obtains zero overhead plus LValue iterators, but if it do not supports it then uses const_reference_pair and obtains minimum overhead with ReadWrite iterators. This might be controversial but the advantages that mutant offers are very big and the truth is that I do not believe that in any actual compiler this idiom is not supported. This scheme would adjust perfectly to the present standard since we are not supposing anything. The only drawback here is that although the mutant approach allows to make LValue iterators we have to degrade they to Read Write in both cases, because we want that the same code can be compiled in any standard compliant compiler.

```
- Hopefully we find our way out of the problem -
```

Joaquin

Changing the subject, I believe that the general concept of hooking data is good, but I do not like the way you implement it. It has to be easy to migrate to B.MI to anticipate the case in that Boost.Bimap becomes insufficient. It is more natural for a B.MI user that the data is accessed without the indirection of `.data`*. I do not know how this can be articulated in your framework.*

Matias

I have a technical problem to implement the data_hook in this way. If the standard would let us use the mutant idiom directly, I can implement it using multiple inheritance. But as we must use const_reference_pair too, It becomes impossible for me to support it. We have three options here:

1) relation { left, right, data } and pair_view { first, second, data }

- This is more intuitive within the bimap framework, since it does not mix the data with the index, as a table in a data base does, but gives more importance to the index.

- It is not necessary that the user puts the mutable keyword in each member of the data class.

- This moves away just a little bit from B.MI because the model of it is similar to a table, but it continues to exist a clear path of migration.

2) relation { left,right, d1,d2... dn } and pair_view { first, second, data }

- The path to B.MI is the one you have proposed.

- It is very asymmetric. It is necessary to explain that the views are handled different that the relation.

- The user must place the mutable keyboards in the data class.

3) Only relation { left,right, d1,d2... dn }

- Simple migration path to B.MI.

- You are not able to access the hooked data from the views.

My vote goes to the first proposal.

Joaquin

Yes, the first option is the one that less surprises hold to the user. I also vote for 1.

- The third week was over -

Matias

There is still one problem that I have to solve. I need to know if it is necessary to create a map_view associated to nothing. If it is necessary there are two options: that it behaves as an empty container or that it throws an exception or assert when trying to use it. If it is not necessary, the map_view is going to keep a reference instead of a pointer. To me, the map_view always must be viewing something. In the case of the iterators being able to create them empty, makes them easy to use in contexts that require constructors by default, like being the value_type of a container, but I do not believe that this is the case of map_view.

Joaquin

How would an empty map_view be useful? My intuition is like yours, map_view would have to be always associate to something. If we wished to obtain the semantics "is associated or not" we can use a pointer to a map_view.

Matias

OK, then you agree to that map_views stores a reference instead of a pointer?

Joaquin

It depends on the semantics you want to give to map_views, and in concrete to the copy of map_views.

```
map_view x=...;
map_view y=...;
x=y;
```

What is supposed to do this last line?

1. Rebinding of x, that is to say, x points at the same container that y.

2. Copy of the underlying container.

If you want to implement 1, you cannot use references internally. If you want to implement 2, it is almost the same to use a reference or a pointer.

Matias

If I want that they behave exactly as std::maps then I must go for 2. But if I think they as "views" of something, I like 1. The question is complicated. I add another option:

3. Error: operator= is declare as private in boost::bimap::map_view std_container

Also What happens with `std_container = view;` *? and with* `view = std_container;` *?*

History

The long path from Code Project to Boost

2002 - bimap at Code Project	Joaquin Lopez Muñoz posted his first bimap library in 2002. Tons of users have been using it. He then asked the list for interest in his library in 2003. Luckily, there was a lot of interest and Joaquin started to boostify the code. At some point all the developers seemed to agree that, rather than a bidirectional map, it would be better to work on an N-indexed set that contained Joaquin's library as a particular case.
2003 - multiindex_set	The library grew enormously and was ready for a formal review in 2003. At this point, the container was a lot more powerful, but everything comes with a price and this new beast lacked the simplicity of the original bimap.
2004 - indexed_set	In 2004, the formal review ended well for the new multi-indexed container. This Swiss army knife introduced several new features, such as non-unique indexes, hashed indices and sequenced indices. In the list of improvements to the library, it was mentioned that a bidirectional map should be coded in top of this container.
2005 - multi_index_container	Once in Boost, the library switched to the now familiar name "Boost.MultiIndex". Late in 2004, it formally became a member of Boost. Joaquin continued to enchance the library and added new features such as composite keys and random-access indices.
2006 - Multi Index Specialized Containers SoC project	In 2006, during the formal review of Boost.Property_tree, the need for a bidirectional map container built on top of Boost.MultiIndex arose again. Boost entered the Google SoC 2006 as a mentor organization at the same time. Joaquin put himself forward as a mentor. He proposed to build not only a bidirectional map, but a myriad multi-indexed specialized containers. Matias Capeletto presented an application to code Boost.Misc for the SoC and was elected, along with nine other students. Matias's and Joaquin's SoC project ends with a working implementation of the bimap library that was presented in an informal review. By the end of the year the library was queued for a formal review.
2007 - Boost.Bimap	The formal review took place at the beggining of the year and Boost.Bimap was accepted in Boost.

MultiIndex and Bimap

This is the conversation thread that began during Boost.PropertyTree formal review process. The review was very interesting and very deep topics were addressed. It is quite interesting and it is now part of this library history. Enjoy!

Marcin

> *The biggest virtue of property_tree is easy to use interface. If we try to make generic tree of it, it will be compromised.*

Gennadiy

> *IMO the same result (as library presents) could be achieved just by using multi_index.*

Marcin

> *Could you elaborate more on that? I considered use of multi_index to implement indexing for properties, but it only affected the implementation part of library, not interface, and because I already had a working, exception safe solution, I didn't see the reason to dump it and add another dependency on another library.*

Gennadiy

> *I mean why do I need this half baked property_tree as another data structure? Property tree supports nothing in itself. It's just a data structure. You have parsers that produce property tree out of different sources. But you mat*

as well produce maps or something else. Here for example All that I need to do to "implement" similar functionality as your property tree:

```cpp
// Data structure itself
template<typename ValueType,typename KeyType>
struct Node;
template<typename ValueType,typename KeyType>
struct ptree_gen {
    typedef std::pair<KeyType,Node<ValueType,KeyType> > mi_value;
    typedef multi_index_container<mi_value, indexed_by<...> > type;
};
template<typename ValueType,typename KeyType>
struct Node {
    ValueType v;
    ptree_gen<ValueType,KeyType>::type children;
};
// serialization support
template<class Archive,typename ValueType,typename KeyType>
void serialize(Archive & ar, Node<ValueType,KeyType>& n,
               const unsigned int version)
{
    ar & n.v;
    ar & n.children;
}
// some access methods
template<typename ValueType,typename KeyType>
ValueType const&
get( string const& keys, ptree_gen<ValueType,KeyType>::type const& src )
{
    std::pait<string,string> sk = split( keys, "." );
    Node const& N = src.find( sk.first );
    return sk.second.empty() ? N.v : get( sk.second, N.children );
}
```

Use it like this:

```cpp
ptree_gen<string,string>::type PT;
boost::archive::text_iarchive ia( std::ifstream ifs("filename") );
ia >> PT;
string value = get( "a.b.c.d", PT );
```

Now tell me how property_tree interface is easier? And what is the value in 50k of Code you need to implement this data structure.

Thorsten

Seriously Gennadiy, do you really see newbies writing the code you just did?

Marcin

What you just implemented is stripped down, bare bones version of property_tree that, among other things, does not allow you to produce human editable XML files. Now add more interface (aka get functions), add more archives to serialization lib, add customization, add transparent translation from strings to arbitrary types and vice versa. Spend some weeks trying to get all the corner cases right, and then some more weeks trying to smooth rough edges in the interface. Then write tests. Write docs. At the end, I believe you will not get much less code than there is in the library already. Maybe you get some savings by using multi_index instead of manual indexing.

The reason why ptree does not use multi index is because implementation existed long before I considered submitting to boost, probably before even I knew of multi index existence. It was working well. Later, when I was improving

it during pre-review process, I seriously considered using multi-index. But I decided it is not worth throwing everything out.

Although ptree has large interface with many functions modifying state of the tree, it uses "single point of change" approach. Every insert eventually goes through one function, which takes care of exception safety and keeping index in sync with data. The same applies to erase. This function has 9 lines of code in case of insert, and (by co-incidence) also 9 in case of erase. By using multi index these functions would obviously be simplified, maybe to 4 lines each. Net gain: 10 lines of code (out of several hundred in ptree_implementation.hpp).

I'm aware that there are performance gains to be reaped as well, but at that time I was rather focusing on getting the interface right.

Dave

That's perfectly reasonable, but (through no fault of yours) it misses the point I was trying to make. I guess I should have said, "...that demonstrates it to be the best implementation."

All I'm saying is that the extent to which a Boost library implementation should leverage other Boost libraries is not a question that can always be decided based on following simple guidelines, and that if this library is accepted, it's worth revisiting your decision.

Thorsten

I think it is important to focus on the interface in the review, but I also see several benefits of an implementation that builds on Boost.MultiIndex:'

- fewer bugs like the one Joaquin found

- better space efficiency

- exception-safety guarantees are immediately full-filled (I haven't looked, but I suspect that there are several bugs in this area)

Daniel

Multi_index supports everything a bimap would, but its interface is more cumbersome. I for one won't use a W3DOM-like library if we get one, but I would happily use property_tree. I've also only used multi_index once, and that was to use it as a bidirectional map. Property_tree covers other areas as well as being a potential subset of an XML library, but I still hold there is value in such a subset.

Boris

I haven't used program_options yet. But if I understand correctly both libraries seem to support storing and accessing data with strings that might describe some kind of hierarchy. This seems to be the core idea of both libraries - is this correct?

Then it wouldn't matter much what container is used. However a generic tree which can store data hierarchically probably makes most sense. If I understand correctly both libraries could make use of such a class?

Marcin

I think generic tree container is material for another library. Whether property_tree should be based on it or not is a matter of internal implementation, and generally of little interest to users. The biggest value of property_tree is in its easy to use interface, that should not be compromised, if at all possible. I have been already reassured in this view by quite many people who took their time to review the library.

Boris

I was trying to see the big picture: I rather prefer a C++ standard based on a few well-known concepts like containers, iterators, algorithms etc. instead of having a C++ standard with hundreds of components which are

tailored for specific needs, collaborate with only a handful of other components and think they provide an easy-to-use interface while all the easy-to-use interfaces make the whole standard less easy-to-use.

That said I have used your property tree library myself to read and write a configuration file. It was indeed very easy to use. However it would have been even easier if it was something I had known before like eg. an iterator. For now I will definitely use your property tree library but would appreciate if existing concepts were reused many C++ developers are familiar with. My opinion is that your library should be a part of Boost but should be more generalized in the future.

Thorsten

Well, I think we need both. Boost.MultiIndex is a great library and can do all kinds of wonderful things. But I would still like to see a bidirectional map (boost::bimap) written as a wrapper around it to get an easy and specialized interface.

Pavel

Bimap is available in libs/multi-index/examples/bimap.cpp.

Thorsten

Right, but the real value comes when somebody designs a nice STL-like interface and write docs etc, at least that was my point.

Dave

IMO Thorsten is exactly right. This is precisely the sort of thing that could be added to the library as part of its ongoing maintenance and development (without review, of course).

Joaquin

Thorsten, we have talked about this privately in the past, but I feel like bringing it to the list in the hope of getting the attention of potential contributors:

There are some data structures buildable with B.MI which are regarded as particularly useful or common, like for instance the bidirectional map or bimap. A lean and mean implementation is provided in the aforementioned example, but certainly a much carefully crafted interface can be provided keeping B.MI as the implementation core: operator[], selection of 1-1/1-N/N-1/N-N variants, hashing/ordering, etc.

I'm afraid I don't have the time to pursue this, as the current roadmap for core features of B.MI is taking all the spare time I can dedicate to the library. For this reason, I would love to see some volunteer jumping in who can develop this and other singular containers using B.MI (a cache container comes to mind) and then propose the results here either as a stand alone library of as part of B.MI --I'd prefer the former so as to keep the size of B.MI bounded.

If there's such a volunteer I can provide her with some help/mentoring. I also wonder whether this is a task suitable to be proposed for Google Summer of Code.

Thorsten

I think it would be good for SOC. All the really hard things are taken care of by B.MI, and so it seems reasonable for a student to be able to fill in the details.

Dave

Great!

Jeff

Please write a proposal!

Joaquin

I've just done so:

Specialized containers with Boost.MultiIndex

Introduction

Boost.MultiIndex allows the construction of complex data structures involving two or more indexing mechanisms on the same set of elements. Out of the unlimited range of possible data structures specifiable within Boost.MultiIndex, some particular configurations arise recurrently:

a. A bidirectional map or bimap is a container of elements of type pair<T,Q> where fast look up is provided both for the T and the Q field, in contrast with a regular STL map which only allows for fast look up on T.

b. An MRU (most recently used) list keeps the n last referenced elements: when a new item is inserted and the list has reached its maximum length, the oldest element is erased, whereas if an insertion is tried of a preexistence element, this gets promoted to the first position. MRU lists can be used to implement dynamic caches and the kind of behavior exhibited by programs featuring a "Recent files" menu command, for instance.

Although Boost.MultiIndex provides the mechanisms to build these common structures, the resulting interface can be cumbersome and too general in comparison with specialized containers focusing on such particular structures.

Goal

To write a library of specialized containers like the ones described above, using Boost.MultiIndex as the implementation core. Besides bimap and MRU list, the student can also propose other specialized containers of interest in the community. It is expected that the library meets the standards of quality required by Boost for an eventual inclusion in this project, which implies a strong emphasis on interface design, documentation and unit testing; the mentor will be guiding the student through the complete cycle from specification and requirements gathering to documentation and actual coding. The final result of the project must then contain:

a. Source code following Boost programming guidelines.

b. User documentation. Requirements on the format are loose, though the QuickBook format is gaining acceptance within Boost.

c. Complete set of unit tests powered by Boost Build System V2.

Requirements

a. Intermediate-to-high level in C++, with emphasis in generic programming (templates).

b. Knowledge of the STL framework and design principles. Of course, knowledge of Boost in general and Boost.MultiIndex in particular is a big plus.

c. Acquaintance with at least two different C++ programming environments.

d. Some fluency in the English language; subsequent reviews of the documentation can help smooth rough edges here, though.

e. A mathematical inclination and previous exposure to a formal Algorithms course would help very much.

f. A craving for extreme quality work.

Benefits for the student

The student taking on this project will have the opportunity to learn the complete process of software production inside a highly regarded C++ open source institution, and even see her work included in Boost eventually. The completion of the project involves non-trivial problems in C++ interface design and so-called modern C++ programming, high quality user documentation and unit testing. The student will also learn, perhaps to her surprise, that most of the time will be spent gathering and trying ideas and, in general, thinking, rather than writing actual code.

Matias

I am planning to submit an application to SoC. I will love to make real the specialized containers you mention and try to include some useful others.

```
And then... after long hours of coding (and fun) this library saw the light.
```

Acknowledgements

This library was developed in the context of the Google SoC 2006. I first want to thank my mentor, Joaquin, for his friendship during this project. Not only did he help me go through the process of creating this library, but he also did his best so we could have a great time doing it. Also, Boost.Bimap would not exist had Boost.MultiIndex, Joaquin's masterpiece, not existed. Thanks a lot!

I want to thank Google for this amazing *boost* to the open-source community and to Boost mentors for trusting in my proposal in the first place. Next on the list are my colleagues from SoC that helped me not get bored during the long hours of coding.

Special acknowledgements to the developers of the Boost libraries that Boost.Bimap has abused. See the dependencies section for a complete list.

I want to thank the open-source developers who wrote the tools I used during this project. The list of names is infinitely long, so I give a general huge thanks here.

Thanks to Paul Giaccone for proof-reading this documentation. (He has not finished yet -- the remaining typos and spelling errors are mine and will be corrected as soon as possible.)

Finally, thanks to my family, who had to see me at home all day during the SoC. Special thanks to my brother Agustin, future famous novelist (at the present time he is 19 years old), who patiently read every word of these docs and while correcting them, barked at me for my bad written English. I have learned a lot from his sermons. I want to thank my dog, Mafalda, too for barking all day from my window and for being such a good company.

Thanks to Alisdair Meredith, Fernando Cacciola, Jeff Garland, John Maddock, Thorsten Ottosen, Tony and Giovanni Piero Deretta for participating in the formal review and give me useful advices to improve this library. And thanks a lot to Ion Gaztañaga for managing the review.

Boost.Bimap Team

From Argentina... Matias and Mafalda and from Spain... Joaquin and Hector
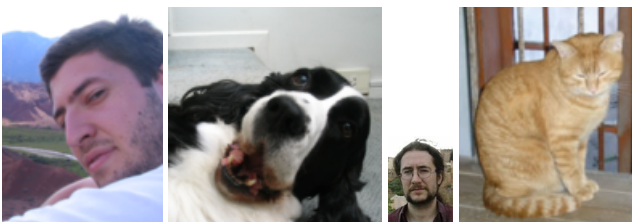
Luckily, the distance helps team members avoid eating each other.