Boost.Icl

Joachim Faulhaber

Table of Contents

Introduction

“A bug crawls across the boost docs on my laptop screen. Let him be! We need all the readers we can get.” -- Freely adapted from Jack Kornfield

Intervals are almost ubiquitous in software development. Yet they are very easily coded into user defined classes by a pair of numbers so they are only *implicitly* used most of the time. The meaning of an interval is simple. They represent all the elements between their lower and upper bound and thus a set. But unlike sets, intervals usually can not be added to a single new interval. If you want to add intervals to a collection of intervals that does still represent a *set*, you arrive at the idea of *interval_sets* provided by this library.

Interval containers of the **ICL** have been developed initially at Cortex Software GmbH to solve problems related to date and time interval computations in the context of a Hospital Information System. Time intervals with associated values like *amount of invoice* or *set of therapies* had to be manipulated in statistics, billing programs and therapy scheduling programs. So the **ICL** emerged out of those industrial use cases. It extracts generic code that helps to solve common problems from the date and time problem domain and can be beneficial in other fields as well.

One of the most advantageous aspects of interval containers is their very compact representation of sets and maps. Working with sets and maps *of elements* can be very inefficient, if in a given problem domain, elements are typically occurring in contiguous chunks. Besides a compact representation of associative containers, that can reduce the cost of space and time drastically, the ICL comes with a universal mechanism of aggregation, that allows to combine associated values in meaningful ways when intervals overlap on insertion.

For a condensed introduction and overview you may want to look at the presentation material on the **ICL** from ***BoostCon2009***.

Definition and Basic Example

The **Interval Container Library (ICL)** provides `intervals` and two kinds of interval containers: `interval_sets` and `interval_maps`.

- An `interval_set` is a **set** that is implemented as a set of intervals.
- An `interval_map` is a **map** that is implemented as a map of interval value pairs.

Two Aspects

`Interval_sets` and `interval_maps` expose two different aspects in their interfaces: (1) The functionality of a set or a map, which is the more ***abstract aspect***. And (2) the functionality of a container of intervals which is the more specific and ***implementation related aspect***. In practice both aspects are useful and are therefore supported.

The first aspect, that will be called ***fundamental aspect***, is the more important one. It means that we can use an `interval_set` or `interval_map` like a set or map ***of elements***. It exposes the same functions.

```
interval_set<int> mySet;
mySet.insert(42);
bool has_answer = contains(mySet, 42);
```

The second aspect, that will be called ***segmental aspect***, allows to exploit the fact, that the elements of `interval_sets` and `interval_maps` are clustered in ***intervals*** or ***segments*** that we can iterate over.

```
// Switch on my favorite telecasts using an interval_set
interval<seconds>::type news(make_seconds("20:00:00"), make_seconds("20:15:00"));
interval<seconds>::type talk_show(make_seconds("22:45:30"), make_seconds("23:30:50"));
interval_set<seconds> myTvProgram;
myTvProgram.add(news).add(talk_show);

// Iterating over elements (seconds) would be silly ...
for(interval_set<seconds>::iterator telecast = myTvProgram.begin();
    telecast != myTvProgram.end(); ++telecast)
    //...so this iterates over intervals
   TV.switch_on(*telecast);
```

Working with `interval_sets` and `interval_maps` can be beneficial whenever the elements of sets appear in contiguous chunks: `intervals`. This is obviously the case in many problem domains, particularly in fields that deal with computations related to date and time.

Addabitlity and Subtractability

Unlike `std::sets` and `maps`, `interval_sets` and `interval_maps` implement concept `Addable` and `Subtractable`. So `interval_sets` define an `operator +=` that is naturally implemented as ***set union*** and an `operator -=` that is consequently implemented as ***set difference***. In the **Icl** `interval_maps` are addable and subtractable as well. It turned out to be a very fruitful concept to propagate the addition or subtraction to the `interval_map's` associated values in cases where the insertion of an interval value pair into an `interval_map` resulted in a collision of the inserted interval value pair with interval value pairs, that are already in the `interval_map`. This operation propagation is called ***aggregate on overlap***.

Aggregate on Overlap

This is a first motivating example of a very small party, demonstrating the ***aggregate on overlap*** principle on `interval_maps`:

In the example Mary enters the party first. She attends during the time interval `[20:00,22:00)`. Harry enters later. He stays within `[21:00,23:00)`.

```
typedef std::set<string> guests;
interval_map<time, guests> party;
party += make_pair(interval<time>::right_open(time("20:00"), time("22:00")), guests("Mary"));
party += make_pair(interval<time>::right_open(time("21:00"), time("23:00")), guests("Harry"));
// party now contains
[20:00, 21:00)->{"Mary"}
[21:00, 22:00)->{"Harry","Mary"} //guest sets aggregated on overlap
[22:00, 23:00)->{"Harry"}
```

On overlap of intervals, the corresponding name sets are ***accumulated***. At the ***points of overlap*** the intervals are ***split***. The accumulation of content on overlap of intervals is done via an `operator +=` that has to be implemented for the associated values of the `interval_map`. In the example the associated values are `guest sets`. Thus a `guest set` has to implement `operator +=` as set union.

As can be seen from the example an `interval_map` has both a ***decompositional behavior*** (on the time dimension) as well as an ***accumulative one*** (on the associated values).

Addability and aggregate on overlap are useful features on `interval_maps` implemented via function `add` and `operator +=`. But you can also use them with the *traditional* insert semantics that behaves like `std::map::insert` generalized for interval insertion.

Icl's class templates

In addition to interval containers we can work with containers of elements that are ***behavioral equal*** to the interval containers: On the fundamental aspect they have exactly the same functionality. An `std::set` of the STL is such an equivalent set implementation. Due to the aggregation facilities of the icl's interval maps `std::map` is fundamentally not completely equivalent to an `interval_map`. Therefore there is an extra `icl::map` class template for maps of elements in the icl.

- The `std::set` is behavioral equal to `interval_sets` on the ***fundamental*** aspect.
- An `icl::map` is behavioral equal to `interval_maps` on the ***fundamental*** aspect. Specifically an `icl::map` implements ***aggregate on overlap***, which is named ***aggregate on collision*** for an element container.

The following tables give an overview over the main class templates provided by the **icl**.

Table 1. Interval class templates

group	form	template
statically bounded	asymmetric	`right_open_interval`
		`left_open_interval`
	symmetric	`closed_interval`
		`open_interval`
dynamically bounded		`discrete_interval`
		`continuous_interval`

Statically bounded intervals always have the same kind of interval borders, e.g. right open borders `[a..b)` for `right_open_interval`. Dynamically bounded intervals can have different borders. Refer to the chapter about ***intervals*** for details.

Table 2. Container class templates

granularity	style	sets	maps
interval	joining	`interval_set`	`interval_map`
	separating	`separate_interval_set`	
	splitting	`split_interval_set`	`split_interval_map`
element		(`std::set`)	`map`

`Std::set` is placed in paretheses, because it is not a class template of the **ICL**. It can be used as element container though that is behavioral equal to the ICL's interval sets on their fundamental aspect. Column ***style*** refers to the different ways in which interval containers combine added intervals. These ***combining styles*** are described in the next section.

Interval Combining Styles

When we add intervals or interval value pairs to interval containers, the intervals can be added in different ways: Intervals can be joined or split or kept separate. The different interval combining styles are shown by example in the tables below.

Table 3. Interval container's ways to combine intervals

	joining	separating	splitting
set	`interval_set`	`separate_interval_set`	`split_interval_set`
map	`interval_map`		`split_interval_map`
	Intervals are joined on overlap or touch (if associated values are equal).	Intervals are joined on overlap, not on touch.	Intervals are split on overlap. All interval borders are preserved.

Table 4. Interval combining styles by example

	joining	separating	splitting
set	`interval_set` *A*	`separate_interval_set` *B*	`split_interval_set` *C*

	`{[1      3)      }`
	`+     [2      4)`
	`+            [4 5)`
	`= {[1           5)}`
		`{[1      3)}     `
		`+     [2      4)`
		`+            [4 5)`
		`= {[1         4)[4 5)}`
			`{[1      3)      }`
			`+     [2      4)`
			`+            [4 5)`
			`= {[1 2)[2 3)[3 4)[4 5)}`

map	`interval_map` *D*		`split_interval_map` *E*

	`{[1      3)->1       }`
	`+     [2      4)->1`
	`+            [4 5)->1`
	`= {[1 2)[2 3)[3    5)   }`
	`   | ->1  ->2   ->1   |`
			`{[1      3)->1       }`
			`+     [2      4)->1`
			`+            [4 5)->1`
			`= {[1 2)[2 3)[3 4)[4 5) }`
			`   | ->1  ->2  ->1  ->1  |`

Note that `interval_sets` *A*, *B* and *C* represent the same set of elements `{1,2,3,4}` and `interval_maps` *D* and *E* represent the same map of elements `{1->1, 2->2, 3->1, 4->1}`. See example program Interval container for an additional demo.

Joining interval containers

`Interval_set` and `interval_map` are always in a ***minimal representation***. This is useful in many cases, where the points of insertion or intersection of intervals are not relevant. So in most instances `interval_set` and `interval_map` will be the first choice for an interval container.

Splitting interval containers

`Split_interval_set` and `split_interval_map` on the contrary have an ***insertion memory***. They do accumulate interval borders both from additions and intersections. This is specifically useful, if we want to enrich an interval container with certain time grids, like e.g. months or calendar weeks or both. See example time grids for months and weeks.

Separating interval containers

`Separate_interval_set` implements the separating style. This style preserves borders, that are never passed by an overlapping interval. So if all intervals that are inserted into a `separate_interval_set` are generated form a certain grid that never pass say month borders, then these borders are preserved in the `separate_interval_set`.

Examples

Overview

Table 5. Overview over Icl Examples

level	example	classes	features
intro	Party	`interval_map`	Generates an attendance history of a party by inserting into an `interval_map`. Demonstrating ***aggregate on overlap***.
basic	Interval	`discrete_interval`, `continuous_interval`	Intervals for discrete and continuous instance types. Closed and open interval borders.
basic	Dynamic intervals	`discrete_interval`, `continuous_interval`, `interval`	Intervals with dynamic interval bounds as library default.
basic	Static intervals	`right_open_interval`, `interval`	Intervals with static interval bounds and changing the library default.
basic	Interval container	`interval_set`, `separate_interval_set`, `split_interval_set`, `split_interval_map`, `interval_map`	Basic characteristics of interval containers.
basic	Overlap counter	`interval_map`	The most simple application of an interval map: Counting the overlaps of added intervals.
advanced	Party's height average	`interval_map`	Using *aggregate on overlap* a history of height averages of party guests is computed. Associated values are user defined class objects, that implement an appropriate `operator +=` for the aggregation.
advanced	Party's tallest guests	`interval_map`, `split_interval_map`	Using *aggregate on overlap* the heights of the party's tallest guests are computed. Associated values are aggregated via a maximum functor, that can be chosen as template parameter of an interval_map class template.
advanced	Time grids for months and weeks	`split_interval_set`	Shows how the ***border preserving*** `split_interval_set` can be used to create time partitions where different periodic time intervals overlay each other.

level	example	classes	features
advanced	Man power	`interval_set`, `interval_map`	Set style operations on `interval_sets` and `interval_maps` like union, difference and intersection can be used to obtain calculations in a flexible way. Example **man_power** demonstrates such operations in the process of calculating the available man-power of a company in a given time interval.
advanced	User groups	`interval_map`	Example **user_groups** shows how interval_maps can be unified or intersected to calculate desired information.
and std	Std copy	`interval_map`	Fill interval containers using `std::copy`.
and std	Std transform	`interval_map`, `separate_interval_set`	Fill interval containers from user defined objects using `std::transform`.
customize	Custom interval	`interval_traits`	Use interval containers with your own interval class types.

Party

Example **party** demonstrates the possibilities of an interval map (`interval_map` or `split_interval_map`). An `interval_map` maps intervals to a given content. In this case the content is a set of party guests represented by their name strings.

As time goes by, groups of people join the party and leave later in the evening. So we add a time interval and a name set to the `interval_map` for the attendance of each group of people, that come together and leave together. On every overlap of intervals, the corresponding name sets are accumulated. At the points of overlap the intervals are split. The accumulation of content is done via an operator += that has to be implemented for the content parameter of the `interval_map`. Finally the interval_map contains the history of attendance and all points in time, where the group of party guests changed.

Party demonstrates a principle that we call ***aggregate on overlap***: On insertion a value associated to the interval is aggregated with those values in the interval_map that overlap with the inserted value. There are two behavioral aspects to ***aggregate on overlap***: a ***decompositional behavior*** and an ***accumulative behavior***.

- The ***decompositional behavior*** splits up intervals on the *time dimension* of the interval_map so that the intervals are split whenever associated values change.
- The ***accumulative behavior*** accumulates associated values on every overlap of an insertion for the associated values.

The aggregation function is += by default. Different aggregations can be used, if desired.

```
// The next line includes <boost/date_time/posix_time/posix_time.hpp>
// and a few lines of adapter code.
#include <boost/icl/ptime.hpp>
#include <iostream>
#include <boost/icl/interval_map.hpp>

using namespace std;
using namespace boost::posix_time;
using namespace boost::icl;

// Type set<string> collects the names of party guests. Since std::set is
// a model of the itl's set concept, the concept provides an operator +=
// that performs a set union on overlap of intervals.
typedef std::set<string> GuestSetT;

void boost_party()
{
    GuestSetT mary_harry;
    mary_harry.insert("Mary");
    mary_harry.insert("Harry");

    GuestSetT diana_susan;
    diana_susan.insert("Diana");
    diana_susan.insert("Susan");

    GuestSetT peter;
    peter.insert("Peter");

    // A party is an interval map that maps time intervals to sets of guests
    interval_map<ptime, GuestSetT> party;

    party.add( // add and element
      make_pair(
        interval<ptime>::right_open(
          time_from_string("2008-05-20 19:30"),
          time_from_string("2008-05-20 23:00")),
        mary_harry));

    party += // element addition can also be done via operator +=
      make_pair(
        interval<ptime>::right_open(
          time_from_string("2008-05-20 20:10"),
          time_from_string("2008-05-21 00:00")),
        diana_susan);

    party +=
      make_pair(
        interval<ptime>::right_open(
          time_from_string("2008-05-20 22:15"),
          time_from_string("2008-05-21 00:30")),
        peter);


    interval_map<ptime, GuestSetT>::iterator it = party.begin();
    cout << "----- History of party guests -------------------------\n";
    while(it != party.end())
    {
        interval<ptime>::type when = it->first;
        // Who is at the party within the time interval 'when' ?
        GuestSetT who = (*it++).second;
        cout << when << ": " << who << endl;
    }
```

```
}


int main()
{
    cout << ">>Interval Container Library: Sample boost_party.cpp <<\n";
    cout << "-------------------------------------------------------\n";
    boost_party();
    return 0;
}

// Program output:
/*-----------------------------------------------------------------------------
>>Interval Container Library: Sample boost_party.cpp <<
-------------------------------------------------------
----- History of party guests -------------------------
[2008-May-20 19:30:00, 2008-May-20 20:10:00): Harry Mary
[2008-May-20 20:10:00, 2008-May-20 22:15:00): Diana Harry Mary Susan
[2008-May-20 22:15:00, 2008-May-20 23:00:00): Diana Harry Mary Peter Susan
[2008-May-20 23:00:00, 2008-May-21 00:00:00): Diana Peter Susan
[2008-May-21 00:00:00, 2008-May-21 00:30:00): Peter
-----------------------------------------------------------------------------*/
```

Caution

We are introducing `interval_maps` using an *interval map* **of sets of strings**, because of it's didactic advantages. The party example is used to give an immediate access to the basic ideas of interval maps and *aggregate on overlap*. For real world applications, an interval_map of sets is not necessarily recommended. It has the same efficiency problems as a `std::map` of `std::sets`. There is a big realm though of using interval_maps with numerical and other efficient data types for the associated values.

Interval

Example **interval** shows some basic functionality of `intervals`.

- Different instances of `intervals` for integral (`int, Time`) and continuous types (`double, std::string`) are used.
- The examples uses open and closed intervals bounds.
- Some borderline functions calls on open interval borders are tested e.g.: `interval<double>::rightopen(1/sqrt(2.0), sqrt(2.0)).contains(sqrt(2.0));`

```
#include <iostream>
#include <string>
#include <math.h>

// Dynamically bounded intervals
#include <boost/icl/discrete_interval.hpp>
#include <boost/icl/continuous_interval.hpp>

// Statically bounded intervals
#include <boost/icl/right_open_interval.hpp>
#include <boost/icl/left_open_interval.hpp>
#include <boost/icl/closed_interval.hpp>
#include <boost/icl/open_interval.hpp>

#include "../toytime.hpp"
#include <boost/icl/rational.hpp>

using namespace std;
using namespace boost;
using namespace boost::icl;

int main()
{
    cout << ">>Interval Container Library: Sample interval.cpp <<\n";
    cout << "----------------------------------------------------\n";

    // Class template discrete_interval can be used for discrete data types
    // like integers, date and time and other types that have a least steppable
    // unit.
    discrete_interval<int>      int_interval
        = construct<discrete_interval<int> >(3, 7, interval_bounds::closed());

    // Class template continuous_interval can be used for continuous data types
    // like double, boost::rational or strings.
    continuous_interval<double> sqrt_interval
        = construct<continuous_interval<double> >(1/sqrt(2.0), sqrt(2.0));
                                             //interval_bounds::right_open() is default
    continuous_interval<string> city_interval
        = construct<continuous_interval<string> >("Barcelona", "Boston", inter↵
val_bounds::left_open());

    discrete_interval<Time>     time_interval
        = construct<discrete_interval<Time> >(Time(monday,8,30), Time(monday,17,20),
                                          interval_bounds::open());

    cout << "Dynamically bounded intervals:\n";
    cout << "  discrete_interval<int>:    " << int_interval  << endl;
    cout << "continuous_interval<double>: " << sqrt_interval << " does "
                                        << string(contains(sqrt_interval, sqrt(2.0))?"":"NOT")
                                           << " contain sqrt(2)" << endl;
    cout << "continuous_interval<string>: " << city_interval << " does "
                                       << string(contains(city_interval,"Barcelona")?"":"NOT")
                                           << " contain 'Barcelona'" << endl;
    cout << "continuous_interval<string>: " << city_interval << " does "
                                         << string(contains(city_interval, "Berlin")?"":"NOT")
                                           << " contain 'Berlin'" << endl;
    cout << "  discrete_interval<Time>:   " << time_interval << "\n\n";

    // There are statically bounded interval types with fixed interval borders
    right_open_interval<string>   fix_interval1; // You will probably use one kind of static in↵
tervals
                                                 // right_open_intervals are recommended.
   closed_interval<unsigned int> fix_interval2; // ... static closed, left_open and open intervals
```

```
    left_open_interval<float>     fix_interval3; // are implemented for sake of completeness but
    open_interval<short>           fix_interval4; // are of minor practical importance.

    right_open_interval<rational<int> > range1(rational<int>(0,1),  rational<int>(2,3));
    right_open_interval<rational<int> > range2(rational<int>(1,3),  rational<int>(1,1));

    // This middle third of the unit interval [0,1)
    cout << "Statically bounded interval:\n";
    cout << "right_open_interval<rational<int>>: " << (range1 & range2) << endl;

    return 0;
}

// Program output:

//>>Interval Container Library: Sample interval.cpp <<
//-------------------------------------------------------
//Dynamically bounded intervals
//  discrete_interval<int>:    [3,7]
//continuous_interval<double>: [0.707107,1.41421) does NOT contain sqrt(2)
//continuous_interval<string>: (Barcelona,Boston] does NOT contain 'Barcelona'
//continuous_interval<string>: (Barcelona,Boston] does  contain 'Berlin'
//  discrete_interval<Time>:   (mon:08:30,mon:17:20)
//
//Statically bounded interval
//right_open_interval<rational<int>>: [1/3,2/3)
```

Dynamic interval

```
#include <iostream>
#include <string>
#include <math.h>
#include <boost/type_traits/is_same.hpp>

#include <boost/icl/interval_set.hpp>
#include <boost/icl/split_interval_set.hpp>
// Dynamically bounded intervals 'discrete_interval' and 'continuous_interval'
// are indirectly included via interval containers as library defaults.
#include "../toytime.hpp"
#include <boost/icl/rational.hpp>

using namespace std;
using namespace boost;
using namespace boost::icl;

int main()
{
    cout << ">>Interval Container Library: Sample interval.cpp <<\n";
    cout << "----------------------------------------------------\n";

    // Dynamically bounded intervals are the library default for
    // interval parameters in interval containers.
    BOOST_STATIC_ASSERT((
        boost::is_same< interval_set<int>::interval_type
                      , discrete_interval<int> >::value
                      ));


    BOOST_STATIC_ASSERT((
        boost::is_same< interval_set<float>::interval_type
                      , continuous_interval<float> >::value
                      ));

    // As we can see the library default chooses the appropriate
    // class template instance discrete_interval<T> or continuous_interval<T>
    // dependent on the domain_type T. The library default for intervals
    // is also available via the template 'interval':
    BOOST_STATIC_ASSERT((
        boost::is_same< interval<int>::type
                      , discrete_interval<int> >::value
                      ));

    BOOST_STATIC_ASSERT((
        boost::is_same< interval<float>::type
                      , continuous_interval<float> >::value
                      ));

    // template interval also provides static functions for the four border types

    interval<int>::type    int_interval  = interval<int>::closed(3, 7);
   interval<double>::type sqrt_interval = interval<double>::right_open(1/sqrt(2.0), sqrt(2.0));
    interval<string>::type city_interval = interval<string>::left_open("Barcelona", "Boston");
    interval<Time>::type   time_interval = inter↵
val<Time>::open(Time(monday,8,30), Time(monday,17,20));

    cout << "----- Dynamically bounded intervals ----------------------------------------\n";
    cout << "  discrete_interval<int>   : " << int_interval  << endl;
    cout << "continuous_interval<double>: " << sqrt_interval << " does "
                                          << string(contains(sqrt_interval, sqrt(2.0))?"":"NOT")
```

```
                                                << " contain sqrt(2)" << endl;
    cout << "continuous_interval<string>: " << city_interval << " does "
                                            << string(contains(city_interval,"Barcelona")?"":"NOT")
                                                << " contain 'Barcelona'" << endl;
    cout << "continuous_interval<string>: " << city_interval << " does "
                                              << string(contains(city_interval, "Berlin")?"":"NOT")
                                                << " contain 'Berlin'" << endl;
    cout << "  discrete_interval<Time>  : " << time_interval << "\n\n";

    // Using dynamically bounded intervals allows to apply operations
    // with intervals and also with elements on all interval containers
    // including interval containers of continuous domain types:

    interval<rational<int> >::type unit_interval
        = interval<rational<int> >::right_open(rational<int>(0), rational<int>(1));
    interval_set<rational<int> > unit_set(unit_interval);
    interval_set<rational<int> > ratio_set(unit_set);
    ratio_set -= rational<int>(1,3); // Subtract 1/3 from the set

    cout << "----- Manipulation of single values in continuous sets ---------------------\n";
    cout << "1/3 subtracted from [0..1) : " << ratio_set << endl;
    cout << "The set does " << string(contains(ratio_set, rational<int>(1,3))?"":"NOT")
                                                << " contain '1/3'" << endl;
    ratio_set ^= unit_set;
    cout << "Flipping the holey set     : " << ratio_set << endl;
    cout << "yields the subtracted      :     1/3\n\n";

    // Of course we can use interval types that are different from the
    // library default by explicit instantiation:
    split_interval_set<int, std::less, closed_interval<Time> > intuitive_times;
    // Interval set 'intuitive_times' uses statically bounded closed intervals
    intuitive_times += closed_interval<Time>(Time(monday,  9,00), Time(monday, 10,59));
    intuitive_times += closed_interval<Time>(Time(monday, 10,00), Time(monday, 11,59));
    cout << "----- Here we are NOT using the library default for intervals --------------\n";
    cout << intuitive_times << endl;

    return 0;
}

// Program output:
//>>Interval Container Library: Sample interval.cpp <<
//------------------------------------------------------
//----- Dynamically bounded intervals ----------------------------------------
//  discrete_interval<int>   : [3,7]
//continuous_interval<double>: [0.707107,1.41421) does NOT contain sqrt(2)
//continuous_interval<string>: (Barcelona,Boston] does NOT contain 'Barcelona'
//continuous_interval<string>: (Barcelona,Boston] does  contain 'Berlin'
//  discrete_interval<Time>  : (mon:08:30,mon:17:20)
//
//----- Manipulation of single values in continuous sets ---------------------
//1/3 subtracted from [0..1) : {[0/1,1/3)(1/3,1/1)}
//The set does NOT contain '1/3'
//Flipping the holey set     : {[1/3,1/3]}
//yields the subtracted      :     1/3
//
//----- Here we are NOT using the library default for intervals --------------
//{[mon:09:00,mon:09:59][mon:10:00,mon:10:59][mon:11:00,mon:11:59]}
```

Static interval

```
#include <iostream>
#include <string>
#include <math.h>
#include <boost/type_traits/is_same.hpp>

// We can change the library default for the interval types by defining
#define BOOST_ICL_USE_STATIC_BOUNDED_INTERVALS
// prior to other inluces from the icl.
// The interval type that is automatically used with interval
// containers then is the statically bounded right_open_interval.

#include <boost/icl/interval_set.hpp>
#include <boost/icl/split_interval_set.hpp>
// The statically bounded interval type 'right_open_interval'
// is indirectly included via interval containers.


#include "../toytime.hpp"
#include <boost/icl/rational.hpp>

using namespace std;
using namespace boost;
using namespace boost::icl;

int main()
{
    cout << ">> Interval Container Library: Sample static_interval.cpp <<\n";
    cout << "------------------------------------------------------------\n";

    // Statically bounded intervals are the user defined library default for
    // interval parameters in interval containers now.
    BOOST_STATIC_ASSERT((
        boost::is_same< interval_set<int>::interval_type
                      , right_open_interval<int> >::value
                      ));

    BOOST_STATIC_ASSERT((
        boost::is_same< interval_set<float>::interval_type
                      , right_open_interval<float> >::value
                      ));

    // As we can see the library default both for discrete and continuous
    // domain_types T is 'right_open_interval<T>'.
    // The user defined library default for intervals is also available via
    // the template 'interval':
    BOOST_STATIC_ASSERT((
        boost::is_same< interval<int>::type
                      , right_open_interval<int> >::value
                      ));

    // Again we are declaring and initializing the four test intervals that have been used
    // in the example 'interval' and 'dynamic_interval'
    interval<int>::type    int_interval  = interval<int>::right_open(3, 8); // shifted the up↵
per bound
   interval<double>::type sqrt_interval = interval<double>::right_open(1/sqrt(2.0), sqrt(2.0));

    // Interval ("Barcelona", "Boston"] can not be represented because there is no 'steppable ↵
next' on
    // lower bound "Barcelona". Ok. this is a different interval:
    interval<string>::type city_interval = interval<string>::right_open("Barcelona", "Boston");
```

```
    // Toy Time is discrete again so we can transfrom open(Time(monday,8,30), Time(monday,17,20))
    //                                       to right_open(Time(monday,8,31), Time(monday,17,20))
    interval<Time>::type   time_interval = inter↵
val<Time>::right_open(Time(monday,8,31), Time(monday,17,20));

    cout << "----- Statically bounded intervals ----------------------------------------\n";
    cout << "right_open_interval<int>   : " << int_interval  << endl;
    cout << "right_open_interval<double>: " << sqrt_interval << " does "
                                         << string(contains(sqrt_interval, sqrt(2.0))?"":"NOT")
                                            << " contain sqrt(2)" << endl;
    cout << "right_open_interval<string>: " << city_interval << " does "
                                        << string(contains(city_interval,"Barcelona")?"":"NOT")
                                            << " contain 'Barcelona'" << endl;
    cout << "right_open_interval<string>: " << city_interval << " does "
                                          << string(contains(city_interval, "Boston")?"":"NOT")
                                            << " contain 'Boston'" << endl;
    cout << "right_open_interval<Time>  : " << time_interval << "\n\n";

    // Using statically bounded intervals does not allows to apply operations
    // with elements on all interval containers, if their domain_type is continuous.
    // The code that follows is identical to example 'dynamic_interval'. Only 'internally'
    // the library default for the interval template now is 'right_open_interval'
    interval<rational<int> >::type unit_interval
        = interval<rational<int> >::right_open(rational<int>(0), rational<int>(1));
    interval_set<rational<int> > unit_set(unit_interval);
    interval_set<rational<int> > ratio_set(unit_set);
    // ratio_set -= rational<int>(1,3); // This line will not compile, because we can not
                                        // represent a singleton interval as right_open_interval.
    return 0;
}

// Program output:
//>> Interval Container Library: Sample static_interval.cpp <<
//--------------------------------------------------------------
//----- Statically bounded intervals ----------------------------------------
//right_open_interval<int>   : [3,8)
//right_open_interval<double>: [0.707107,1.41421) does NOT contain sqrt(2)
//right_open_interval<string>: [Barcelona,Boston) does  contain 'Barcelona'
//right_open_interval<string>: [Barcelona,Boston) does NOT contain 'Boston'
//right_open_interval<Time>  : [mon:08:31,mon:17:20)
```

Interval container

Example **interval container** demonstrates the characteristic behaviors of different interval containers that are also summarized in the introductory Interval Combining Styles.

```
#include <iostream>
#include <boost/icl/interval_set.hpp>
#include <boost/icl/separate_interval_set.hpp>
#include <boost/icl/split_interval_set.hpp>
#include <boost/icl/split_interval_map.hpp>
#include "../toytime.hpp"

using namespace std;
using namespace boost::icl;

void interval_container_basics()
{
    interval<Time>::type night_and_day(Time(monday,   20,00), Time(tuesday,  20,00));
    interval<Time>::type day_and_night(Time(tuesday,   7,00), Time(wednesday, 7,00));
    interval<Time>::type  next_morning(Time(wednesday, 7,00), Time(wednesday,10,00));
    interval<Time>::type  next_evening(Time(wednesday,18,00), Time(wednesday,21,00));

   // An interval set of type interval_set joins intervals that that overlap or touch each other.
    interval_set<Time> joinedTimes;
    joinedTimes.insert(night_and_day);
    joinedTimes.insert(day_and_night); //overlapping in 'day' [07:00, 20.00)
    joinedTimes.insert(next_morning);  //touching
    joinedTimes.insert(next_evening);  //disjoint

    cout << "Joined times  :" << joinedTimes << endl;

    // A separate interval set of type separate_interval_set joins intervals that that
    // overlap but it preserves interval borders that just touch each other. You may
    // represent time grids like the months of a year as a split_interval_set.
    separate_interval_set<Time> separateTimes;
    separateTimes.insert(night_and_day);
    separateTimes.insert(day_and_night); //overlapping in 'day' [07:00, 20.00)
    separateTimes.insert(next_morning);  //touching
    separateTimes.insert(next_evening);  //disjoint

    cout << "Separate times:" << separateTimes << endl;

    // A split interval set of type split_interval_set preserves all interval
    // borders. On insertion of overlapping intervals the intervals in the
    // set are split up at the interval borders of the inserted interval.
    split_interval_set<Time> splitTimes;
    splitTimes += night_and_day;
    splitTimes += day_and_night; //overlapping in 'day' [07:00, 20:00)
    splitTimes += next_morning;  //touching
    splitTimes += next_evening;  //disjoint

    cout << "Split times   :\n" << splitTimes << endl;

    // A split interval map splits up inserted intervals on overlap and aggregates the
    // associated quantities via the operator +=
    split_interval_map<Time, int> overlapCounter;
    overlapCounter += make_pair(night_and_day,1);
    overlapCounter += make_pair(day_and_night,1); //overlapping in 'day' [07:00, 20.00)
    overlapCounter += make_pair(next_morning, 1); //touching
    overlapCounter += make_pair(next_evening, 1); //disjoint

    cout << "Split times overlap counted:\n" << overlapCounter << endl;

    // An interval map joins touching intervals, if associated values are equal
    interval_map<Time, int> joiningOverlapCounter;
    joiningOverlapCounter = overlapCounter;
    cout << "Times overlap counted:\n" << joiningOverlapCounter << endl;
}
```

```
int main()
{
    cout << ">>Interval Container Library: Sample interval_container.cpp <<\n";
    cout << "--------------------------------------------------------------\n";
    interval_container_basics();
    return 0;
}


// Program output:
/* ----------------------------------------------------------------------------
>>Interval Container Library: Sample interval_container.cpp <<
--------------------------------------------------------------
Joined times  :[mon:20:00,wed:10:00)[wed:18:00,wed:21:00)
Separate times:[mon:20:00,wed:07:00)[wed:07:00,wed:10:00)[wed:18:00,wed:21:00)
Split times   :
[mon:20:00,tue:07:00)[tue:07:00,tue:20:00)[tue:20:00,wed:07:00)
[wed:07:00,wed:10:00)[wed:18:00,wed:21:00)
Split times overlap counted:
{([mon:20:00,tue:07:00)->1)([tue:07:00,tue:20:00)->2)([tue:20:00,wed:07:00)->1)
([wed:07:00,wed:10:00)->1)([wed:18:00,wed:21:00)->1)}
Times overlap counted:
{([mon:20:00,tue:07:00)->1)([tue:07:00,tue:20:00)->2)([tue:20:00,wed:10:00)->1)
([wed:18:00,wed:21:00)->1)}
-----------------------------------------------------------------------------*/
```

Overlap counter

Example **overlap counter** provides the simplest application of an interval_map that maps intervals to integers. An interval_map<int,int> serves as an overlap counter if we only add interval value pairs that carry 1 as associated value.

Doing so, the associated values that are accumulated in the interval_map are just the number of overlaps of all added intervals.

```
#include <iostream>
#include <boost/icl/split_interval_map.hpp>

using namespace std;
using namespace boost::icl;


/* The most simple example of an interval_map is an overlap counter.
   If intervals are added that are associated with the value 1,
   all overlaps of added intervals are counted as a result in the
   associated values.
*/
typedef interval_map<int, int> OverlapCounterT;

void print_overlaps(const OverlapCounterT& counter)
{
    for(OverlapCounterT::const_iterator it = counter.begin(); it != counter.end(); it++)
    {
        discrete_interval<int> itv  = (*it).first;
        int overlaps_count = (*it).second;
        if(overlaps_count == 1)
            cout << "in interval " << itv << " intervals do not overlap" << endl;
        else
          cout << "in interval " << itv << ": "<< overlaps_count << " intervals overlap" << endl;
    }
}

void overlap_counter()
{
    OverlapCounterT overlap_counter;
    discrete_interval<int> inter_val;

    inter_val = discrete_interval<int>::right_open(4,8);
   cout << "-- adding   " << inter_val << " -----------------------------------------" << endl;
    overlap_counter += make_pair(inter_val, 1);
    print_overlaps(overlap_counter);
    cout << "-----------------------------------------------------------" << endl;

    inter_val = discrete_interval<int>::right_open(6,9);
   cout << "-- adding   " << inter_val << " -----------------------------------------" << endl;
    overlap_counter += make_pair(inter_val, 1);
    print_overlaps(overlap_counter);
    cout << "-----------------------------------------------------------" << endl;

    inter_val = discrete_interval<int>::right_open(1,9);
   cout << "-- adding   " << inter_val << " -----------------------------------------" << endl;
    overlap_counter += make_pair(inter_val, 1);
    print_overlaps(overlap_counter);
    cout << "-----------------------------------------------------------" << endl;

}

int main()
{
    cout << ">>Interval Container Library: Sample overlap_counter.cpp <<\n";
    cout << "-----------------------------------------------------------\n";
    overlap_counter();
    return 0;
}

// Program output:

// >>Interval Container Library: Sample overlap_counter.cpp <<
```

```
// ----------------------------------------------------------
// -- adding   [4,8) ----------------------------------------
// in interval [4,8) intervals do not overlap
// ----------------------------------------------------------
// -- adding   [6,9) ----------------------------------------
// in interval [4,6) intervals do not overlap
// in interval [6,8): 2 intervals overlap
// in interval [8,9) intervals do not overlap
// ----------------------------------------------------------
// -- adding   [1,9) ----------------------------------------
// in interval [1,4) intervals do not overlap
// in interval [4,6): 2 intervals overlap
// in interval [6,8): 3 intervals overlap
// in interval [8,9): 2 intervals overlap
// ----------------------------------------------------------
```

Party's height average

In the example `partys_height_average.cpp` we compute yet another aggregation: The average height of guests. This is done by defining a `class counted_sum` that sums up heights and counts the number of guests via an `operator +=`.

Based on the `operator +=` we can aggregate counted sums on addition of interval value pairs into an interval_map.

```
// The next line includes <boost/date_time/posix_time/posix_time.hpp>
// and a few lines of adapter code.
#include <boost/icl/ptime.hpp>
#include <iostream>
#include <boost/icl/interval_map.hpp>
#include <boost/icl/split_interval_map.hpp>

using namespace std;
using namespace boost::posix_time;
using namespace boost::icl;


class counted_sum
{
public:
    counted_sum():_sum(0),_count(0){}
    counted_sum(int sum):_sum(sum),_count(1){}

    int sum()const  {return _sum;}
    int count()const{return _count;}
    double average()const{ return _count==0 ? 0.0 : _sum/static_cast<double>(_count); }

    counted_sum& operator += (const counted_sum& right)
    { _sum += right.sum(); _count += right.count(); return *this; }

private:
    int _sum;
    int _count;
};

bool operator == (const counted_sum& left, const counted_sum& right)
{ return left.sum()==right.sum() && left.count()==right.count(); }


void partys_height_average()
{
    interval_map<ptime, counted_sum> height_sums;

    height_sums +=
      make_pair(
        discrete_interval<ptime>::right_open(
          time_from_string("2008-05-20 19:30"),
          time_from_string("2008-05-20 23:00")),
        counted_sum(165)); // Mary is 1,65 m tall.

    height_sums +=
      make_pair(
        discrete_interval<ptime>::right_open(
          time_from_string("2008-05-20 19:30"),
          time_from_string("2008-05-20 23:00")),
        counted_sum(180)); // Harry is 1,80 m tall.

    height_sums +=
      make_pair(
        discrete_interval<ptime>::right_open(
          time_from_string("2008-05-20 20:10"),
          time_from_string("2008-05-21 00:00")),
        counted_sum(170)); // Diana is 1,70 m tall.

    height_sums +=
      make_pair(
        discrete_interval<ptime>::right_open(
          time_from_string("2008-05-20 20:10"),
```

```
          time_from_string("2008-05-21 00:00")),
        counted_sum(165)); // Susan is 1,65 m tall.

    height_sums +=
      make_pair(
        discrete_interval<ptime>::right_open(
          time_from_string("2008-05-20 22:15"),
          time_from_string("2008-05-21 00:30")),
        counted_sum(200)); // Peters height is 2,00 m

    interval_map<ptime, counted_sum>::iterator height_sum_ = height_sums.begin();
    cout << "-------------- History of average guest height -------------------\n";
    while(height_sum_ != height_sums.end())
    {
        discrete_interval<ptime> when = height_sum_->first;

        double height_average = (*height_sum_++).second.average();
        cout << setprecision(3)
             << "[" << first(when) << " - " << upper(when) << ")"
             << ": " << height_average <<" cm = " << height_average/30.48 << " ft" << endl;
    }
}


int main()
{
    cout << ">>Interval Container Library: Sample partys_height_average.cpp  <<\n";
    cout << "-------------------------------------------------------------------\n";
    partys_height_average();
    return 0;
}

// Program output:
/*-----------------------------------------------------------------------------
>>Interval Container Library: Sample partys_height_average.cpp  <<
-------------------------------------------------------------------
-------------- History of average guest height -------------------
[2008-May-20 19:30:00 - 2008-May-20 20:10:00): 173 cm = 5.66 ft
[2008-May-20 20:10:00 - 2008-May-20 22:15:00): 170 cm = 5.58 ft
[2008-May-20 22:15:00 - 2008-May-20 23:00:00): 176 cm = 5.77 ft
[2008-May-20 23:00:00 - 2008-May-21 00:00:00): 178 cm = 5.85 ft
[2008-May-21 00:00:00 - 2008-May-21 00:30:00): 200 cm = 6.56 ft
-----------------------------------------------------------------------------*/
```

Required for `class counted_sum` is a default constructor `counted_sum()`and an `operator ==` to test equality. To enable additive aggregation on overlap also an `operator +=` is needed.

Note that no `operator -=` for a subtraction of `counted_sum` values is defined. So you can only add to the `interval_map<ptime, counted_sum>` but not subtract from it.

In many real world applications only addition is needed and user defined classes will work fine, if they only implement `operator +=`. Only if any of the `operators -=` or `-` is called on the interval_map, the user defined class has to implement it's own `operator -=` to provide the subtractive aggregation on overlap.

Party's tallest guests

Defining `operator +=` (and `-=`) is probably the most important method to implement arbitrary kinds of user defined aggregations. An alternative way to choose a desired aggregation is to instantiate an interval_map class template with an appropriate ***aggregation functor***. For the most common kinds of aggregation the **icl** provides such functors as shown in the example.

Example `partys_tallest_guests.cpp` also demonstrates the difference between an `interval_map` that joins intervals for equal associated values and a `split_interval_map` that preserves all borders of inserted intervals.

```
// The next line includes <boost/date_time/posix_time/posix_time.hpp>
// and a few lines of adapter code.
#include <boost/icl/ptime.hpp>
#include <iostream>
#include <boost/icl/interval_map.hpp>
#include <boost/icl/split_interval_map.hpp>

using namespace std;
using namespace boost::posix_time;
using namespace boost::icl;


// A party's height shall be defined as the maximum height of all guests ;-)
// The last parameter 'inplace_max' is a functor template that calls a max
// aggregation on overlap.
typedef interval_map<ptime, int, partial_absorber, less, inplace_max>
    PartyHeightHistoryT;

// Using a split_interval_map we preserve interval splittings that occurred via insertion.
typedef split_interval_map<ptime, int, partial_absorber, less, inplace_max>
    PartyHeightSplitHistoryT;

void partys_height()
{
    PartyHeightHistoryT tallest_guest;

    tallest_guest +=
      make_pair(
        discrete_interval<ptime>::right_open(
          time_from_string("2008-05-20 19:30"),
          time_from_string("2008-05-20 23:00")),
        180); // Mary & Harry: Harry is 1,80 m tall.

    tallest_guest +=
      make_pair(
        discrete_interval<ptime>::right_open(
          time_from_string("2008-05-20 20:10"),
          time_from_string("2008-05-21 00:00")),
        170); // Diana & Susan: Diana is 1,70 m tall.

    tallest_guest +=
      make_pair(
        discrete_interval<ptime>::right_open(
          time_from_string("2008-05-20 22:15"),
          time_from_string("2008-05-21 00:30")),
        200); // Peters height is 2,00 m

    PartyHeightHistoryT::iterator height_ = tallest_guest.begin();
    cout << "-------------- History of maximum guest height -------------------\n";
    while(height_ != tallest_guest.end())
    {
        discrete_interval<ptime> when = height_->first;
        // Of what height are the tallest guests within the time interval 'when' ?
        int height = (*height_++).second;
        cout << "[" << first(when) << " - " << upper(when) << ")"
             << ": " << height <<" cm = " << height/30.48 << " ft" << endl;
    }

}
```

```
// Next we are using a split_interval_map instead of a joining interval_map
void partys_split_height()
{
    PartyHeightSplitHistoryT tallest_guest;

    // adding an element can be done wrt. simple aggregate functions
    // like e.g. min, max etc. in their 'inplace' or op= incarnation
    tallest_guest +=
      make_pair(
        discrete_interval<ptime>::right_open(
          time_from_string("2008-05-20 19:30"),
          time_from_string("2008-05-20 23:00")),
        180); // Mary & Harry: Harry is 1,80 m tall.

    tallest_guest +=
      make_pair(
        discrete_interval<ptime>::right_open(
          time_from_string("2008-05-20 20:10"),
          time_from_string("2008-05-21 00:00")),
        170); // Diana & Susan: Diana is 1,70 m tall.

    tallest_guest +=
      make_pair(
        discrete_interval<ptime>::right_open(
          time_from_string("2008-05-20 22:15"),
          time_from_string("2008-05-21 00:30")),
        200); // Peters height is 2,00 m

    PartyHeightSplitHistoryT::iterator height_ = tallest_guest.begin();
    cout << "\n";
    cout << "-------- Split History of maximum guest height -------------------\n";
    cout << "--- Same map as above but split for every interval insertion.  ---\n";
    while(height_ != tallest_guest.end())
    {
        discrete_interval<ptime> when = height_->first;
        // Of what height are the tallest guests within the time interval 'when' ?
        int height = (*height_++).second;
        cout << "[" << first(when) << " - " << upper(when) << ")"
             << ": " << height <<" cm = " << height/30.48 << " ft" << endl;
    }

}


int main()
{
    cout << ">>Interval Container Library: Sample partys_tallest_guests.cpp  <<\n";
    cout << "------------------------------------------------------------------\n";
    partys_height();
    partys_split_height();
    return 0;
}

// Program output:
/*-----------------------------------------------------------------------------
>>Interval Container Library: Sample partys_tallest_guests.cpp  <<
-------------------------------------------------------------------
-------------- History of maximum guest height -------------------
[2008-May-20 19:30:00 - 2008-May-20 22:15:00): 180 cm = 5.90551 ft
[2008-May-20 22:15:00 - 2008-May-21 00:30:00): 200 cm = 6.56168 ft

-------- Split History of maximum guest height -------------------
--- Same map as above but split for every interval insertion.  ---
```

```
[2008-May-20 19:30:00 - 2008-May-20 20:10:00): 180 cm = 5.90551 ft
[2008-May-20 20:10:00 - 2008-May-20 22:15:00): 180 cm = 5.90551 ft
[2008-May-20 22:15:00 - 2008-May-20 23:00:00): 200 cm = 6.56168 ft
[2008-May-20 23:00:00 - 2008-May-21 00:00:00): 200 cm = 6.56168 ft
[2008-May-21 00:00:00 - 2008-May-21 00:30:00): 200 cm = 6.56168 ft
-----------------------------------------------------------------------------*/
```

Time grids for months and weeks

A `split_interval_set` preserves all interval borders on insertion and intersection operations. So given a `split_interval_set` and an addition of an interval

```
x =  {[1,      3)}
x.add(     [2,      4))
```

then the intervals are split at their borders

```
x == {[1,2)[2,3)[3,4)}
```

Using this property we can intersect `split_interval_maps` in order to iterate over intervals accounting for all occurring changes of interval borders.

In this example we provide an intersection of two `split_interval_sets` representing a month and week time grid.

```
// The next line includes <boost/gregorian/date.hpp>
// and a few lines of adapter code.
#include <boost/icl/gregorian.hpp>
#include <iostream>
#include <boost/icl/split_interval_set.hpp>

using namespace std;
using namespace boost::gregorian;
using namespace boost::icl;

typedef split_interval_set<boost::gregorian::date> date_grid;

// This function splits a gregorian::date interval 'scope' into a month grid:
// For every month contained in 'scope' that month is contained as interval
// in the resulting split_interval_set.
date_grid month_grid(const discrete_interval<date>& scope)
{
    split_interval_set<date> month_grid;

    date frame_months_1st = first(scope).end_of_month() + days(1) - months(1);
    month_iterator month_iter(frame_months_1st);

    for(; month_iter <= last(scope); ++month_iter)
       month_grid += discrete_interval<date>::right_open(*month_iter, *month_iter + months(1));

    month_grid &= scope; // cut off the surplus

    return month_grid;
}

// This function splits a gregorian::date interval 'scope' into a week grid:
// For every week contained in 'scope' that month is contained as interval
// in the resulting split_interval_set.
date_grid week_grid(const discrete_interval<date>& scope)
{
    split_interval_set<date> week_grid;

    date frame_weeks_1st = first(scope) + days(days_until_weekday(first(scope), greg_week↵
day(Monday))) - weeks(1);
    week_iterator week_iter(frame_weeks_1st);

    for(; week_iter <= last(scope); ++week_iter)
       week_grid.insert(discrete_interval<date>::right_open(*week_iter, *week_iter + weeks(1)));

    week_grid &= scope; // cut off the surplus

    return week_grid;
}

// For a period of two months, starting from today, the function
// computes a partitioning for months and weeks using intersection
// operator &= on split_interval_sets.
void month_and_time_grid()
{
    date someday = day_clock::local_day();
    date thenday = someday + months(2);

    discrete_interval<date> itv = discrete_interval<date>::right_open(someday, thenday);

    // Compute a month grid
    date_grid month_and_week_grid = month_grid(itv);
    // Intersection of the month and week grids:
    month_and_week_grid &= week_grid(itv);
```

```
    cout << "interval : " << first(itv) << " - " << last(itv)
         << " month and week partitions:" << endl;
    cout << "---------------------------------------------------------------\n";

    for(date_grid::iterator it = month_and_week_grid.begin();
        it != month_and_week_grid.end(); it++)
    {
        if(first(*it).day() == 1)
            cout << "new month: ";
        else if(first(*it).day_of_week()==greg_weekday(Monday))
            cout << "new week : " ;
        else if(it == month_and_week_grid.begin())
            cout << "first day: " ;
        cout << first(*it) << " - " << last(*it) << endl;
    }
}


int main()
{
    cout << ">>Interval Container Library: Sample month_and_time_grid.cpp <<\n";
    cout << "---------------------------------------------------------------\n";
    month_and_time_grid();
    return 0;
}

// Program output:
/*
>>Interval Container Library: Sample month_and_time_grid.cpp <<
---------------------------------------------------------------
interval : 2008-Jun-22 - 2008-Aug-21 month and week partitions:
---------------------------------------------------------------
first day: 2008-Jun-22 - 2008-Jun-22
new week : 2008-Jun-23 - 2008-Jun-29
new week : 2008-Jun-30 - 2008-Jun-30
new month: 2008-Jul-01 - 2008-Jul-06
new week : 2008-Jul-07 - 2008-Jul-13
new week : 2008-Jul-14 - 2008-Jul-20
new week : 2008-Jul-21 - 2008-Jul-27
new week : 2008-Jul-28 - 2008-Jul-31
new month: 2008-Aug-01 - 2008-Aug-03
new week : 2008-Aug-04 - 2008-Aug-10
new week : 2008-Aug-11 - 2008-Aug-17
new week : 2008-Aug-18 - 2008-Aug-21
*/
```

Man power

`Interval_sets` and `interval_maps` can be filled and manipulated using set style operations such as union +=, difference -= and intersection &=.

In this example **man power** a number of those operations are demonstrated in the process of calculation the available working times (man-power) of a company's employees accounting for weekends, holidays, sickness times and vacations.

```
// The next line includes <boost/gregorian/date.hpp>
// and a few lines of adapter code.
#include <boost/icl/gregorian.hpp>
#include <iostream>
#include <boost/icl/discrete_interval.hpp>
#include <boost/icl/interval_map.hpp>

using namespace std;
using namespace boost::gregorian;
using namespace boost::icl;


// Function weekends returns the interval_set of weekends that are contained in
// the date interval 'scope'
interval_set<date> weekends(const discrete_interval<date>& scope)
{
    interval_set<date> weekends;

    date cur_weekend_sat
        = first(scope)
          + days(days_until_weekday(first(scope), greg_weekday(Saturday)))
          - weeks(1);
    week_iterator week_iter(cur_weekend_sat);

    for(; week_iter <= last(scope); ++week_iter)
        weekends += discrete_interval<date>::right_open(*week_iter, *week_iter + days(2));

    weekends &= scope; // cut off the surplus

    return weekends;
}

// The available working time for the employees of a company is calculated
// for a period of 3 months accounting for weekends and holidays.
//    The available daily working time for the employees is calculated
// using interval_sets and interval_maps demonstrating a number of
// addition, subtraction and intersection operations.
void man_power()
{
    date someday = from_string("2008-08-01");
    date thenday = someday + months(3);

    discrete_interval<date> scope = discrete_interval<date>::right_open(someday, thenday);

    // ---------------------------------------------------------------------
    // (1) In a first step, the regular working times are computed for the
    // company within the given scope. From all available days, the weekends
    // and holidays have to be subtracted:
    interval_set<date> worktime(scope);
    // Subtract the weekends
    worktime -= weekends(scope);
    // Subtract holidays
    worktime -= from_string("2008-10-03"); //German reunification ;)

    // company holidays (fictitious ;)
    worktime -= discrete_interval<date>::closed(from_string("2008-08-18"),
                                                from_string("2008-08-22"));

    //----------------------------------------------------------------------
    // (2) Now we calculate the individual work times for some employees
    //----------------------------------------------------------------------
    // In the company works Claudia.
    // This is the map of her regular working times:
```

```
interval_map<date,int> claudias_working_hours;

// Claudia is working 8 hours a day. So the next statement says
// that every day in the whole scope is mapped to 8 hours worktime.
claudias_working_hours += make_pair(scope, 8);

// But Claudia only works 8 hours on regular working days so we do
// an intersection of the interval_map with the interval_set worktime:
claudias_working_hours &= worktime;

// Yet, in addition Claudia has her own absence times like
discrete_interval<date> claudias_seminar (from_string("2008-09-16"),
                                          from_string("2008-09-24"),
                                          interval_bounds::closed());
discrete_interval<date> claudias_vacation(from_string("2008-08-01"),
                                          from_string("2008-08-14"),
                                          interval_bounds::closed());

interval_set<date> claudias_absence_times(claudias_seminar);
claudias_absence_times += claudias_vacation;

// All the absence times have to subtracted from the map of her working times
claudias_working_hours -= claudias_absence_times;

//-----------------------------------------------------------------------
// Claudia's boss is Bodo. He only works part time.
// This is the map of his regular working times:
interval_map<date,int> bodos_working_hours;

// Bodo is working 4 hours a day.
bodos_working_hours += make_pair(scope, 4);

// Bodo works only on regular working days
bodos_working_hours &= worktime;

// Bodos additional absence times
discrete_interval<date>     bodos_flu(from_string("2008-09-19"), from_string("2008-09-29"),
                                      interval_bounds::closed());
discrete_interval<date> bodos_vacation(from_string("2008-08-15"), from_string("2008-09-03"),
                                      interval_bounds::closed());

interval_set<date> bodos_absence_times(bodos_flu);
bodos_absence_times += bodos_vacation;

// All the absence times have to be subtracted from the map of his working times
bodos_working_hours -= bodos_absence_times;

//-----------------------------------------------------------------------
// (3) Finally we want to calculate the available manpower of the company
// for the selected time scope: This is done by adding up the employees
// working time maps:
interval_map<date,int> manpower;
manpower += claudias_working_hours;
manpower += bodos_working_hours;


cout << first(scope) << " - " << last(scope)
     << "    available man-power:" << endl;
cout << "---------------------------------------------------------------\n";

for(interval_map<date,int>::iterator it = manpower.begin();
    it != manpower.end(); it++)
{
```

```
        cout << first(it->first) << " - " << last(it->first)
             << " -> " << it->second << endl;
    }
}

int main()
{
    cout << ">>Interval Container Library: Sample man_power.cpp <<\n";
    cout << "---------------------------------------------------------------\n";
    man_power();
    return 0;
}

// Program output:
/*
>>Interval Container Library: Sample man_power.cpp <<
---------------------------------------------------------------
2008-Aug-01 - 2008-Oct-31    available man-power:
---------------------------------------------------------------
2008-Aug-01 - 2008-Aug-01 -> 4
2008-Aug-04 - 2008-Aug-08 -> 4
2008-Aug-11 - 2008-Aug-14 -> 4
2008-Aug-15 - 2008-Aug-15 -> 8
2008-Aug-25 - 2008-Aug-29 -> 8
2008-Sep-01 - 2008-Sep-03 -> 8
2008-Sep-04 - 2008-Sep-05 -> 12
2008-Sep-08 - 2008-Sep-12 -> 12
2008-Sep-15 - 2008-Sep-15 -> 12
2008-Sep-16 - 2008-Sep-18 -> 4
2008-Sep-25 - 2008-Sep-26 -> 8
2008-Sep-29 - 2008-Sep-29 -> 8
2008-Sep-30 - 2008-Oct-02 -> 12
2008-Oct-06 - 2008-Oct-10 -> 12
2008-Oct-13 - 2008-Oct-17 -> 12
2008-Oct-20 - 2008-Oct-24 -> 12
2008-Oct-27 - 2008-Oct-31 -> 12
*/
```

User groups

Example **user groups** shows the availability of set operations on `interval_maps`.

In the example there is a user group `med_users` of a hospital staff that has the authorisation to handle medical data of patients. User group `admin_users` has access to administrative data like health insurance and financial data.

The membership for each user in one of the user groups has a time interval of validity. The group membership begins and ends.

- Using a union operation `+` we can have an overview over the unified user groups and the membership dates of employees.
- Computing an intersection `&` shows who is member of both med_users and admin_users at what times.

```
// The next line includes <boost/gregorian/date.hpp>
// and a few lines of adapter code.
#include <boost/icl/gregorian.hpp>
#include <iostream>
#include <boost/icl/interval_map.hpp>

using namespace std;
using namespace boost::gregorian;
using namespace boost::icl;

// Type icl::set<string> collects the names a user group's members. Therefore
// it needs to implement operator += that performs a set union on overlap of
// intervals.
typedef std::set<string> MemberSetT;

// boost::gregorian::date is the domain type the interval map.
// It's key values are therefore time intervals: discrete_interval<date>. The content
// is the set of names: MemberSetT.
typedef interval_map<date, MemberSetT> MembershipT;

// Collect user groups for medical and administrative staff and perform
// union and intersection operations on the collected membership schedules.
void user_groups()
{
    MemberSetT mary_harry;
    mary_harry.insert("Mary");
    mary_harry.insert("Harry");

    MemberSetT diana_susan;
    diana_susan.insert("Diana");
    diana_susan.insert("Susan");

    MemberSetT chief_physician;
    chief_physician.insert("Dr.Jekyll");

    MemberSetT director_of_admin;
    director_of_admin.insert("Mr.Hyde");

    //----- Collecting members of user group: med_users -------------------
    MembershipT med_users;

    med_users.add( // add and element
      make_pair(
        discrete_interval<date>::closed(
          from_string("2008-01-01"), from_string("2008-12-31")), mary_harry));

    med_users +=  // element addition can also be done via operator +=
      make_pair(
        discrete_interval<date>::closed(
          from_string("2008-01-15"), from_string("2008-12-31")),
          chief_physician);

    med_users +=
      make_pair(
        discrete_interval<date>::closed(
          from_string("2008-02-01"), from_string("2008-10-15")),
          director_of_admin);

    //----- Collecting members of user group: admin_users ------------------
    MembershipT admin_users;

    admin_users += // element addition can also be done via operator +=
      make_pair(
```

```
    discrete_interval<date>::closed(
      from_string("2008-03-20"), from_string("2008-09-30")), diana_susan);

admin_users +=
  make_pair(
    discrete_interval<date>::closed(
      from_string("2008-01-15"), from_string("2008-12-31")),
      chief_physician);

admin_users +=
  make_pair(
    discrete_interval<date>::closed(
      from_string("2008-02-01"), from_string("2008-10-15")),
      director_of_admin);

MembershipT all_users   = med_users + admin_users;

MembershipT super_users = med_users & admin_users;

MembershipT::iterator med_ = med_users.begin();
cout << "----- Membership of medical staff -----------------------------------\n";
while(med_ != med_users.end())
{
    discrete_interval<date> when = (*med_).first;
    // Who is member of group med_users within the time interval 'when' ?
    MemberSetT who = (*med_++).second;
    cout << "[" << first(when) << " - " << last(when) << "]"
         << ": " << who << endl;
}

MembershipT::iterator admin_ = admin_users.begin();
cout << "----- Membership of admin staff -------------------------------------\n";
while(admin_ != admin_users.end())
{
    discrete_interval<date> when = (*admin_).first;
    // Who is member of group admin_users within the time interval 'when' ?
    MemberSetT who = (*admin_++).second;
    cout << "[" << first(when) << " - " << last(when) << "]"
         << ": " << who << endl;
}

MembershipT::iterator all_ = all_users.begin();
cout << "----- Membership of all users (med + admin) -------------------------\n";
while(all_ != all_users.end())
{
    discrete_interval<date> when = (*all_).first;
    // Who is member of group med_users OR admin_users ?
    MemberSetT who = (*all_++).second;
    cout << "[" << first(when) << " - " << last(when) << "]"
         << ": " << who << endl;
}

MembershipT::iterator super_ = super_users.begin();
cout << "----- Membership of super users: intersection(med,admin) ------------\n";
while(super_ != super_users.end())
{
    discrete_interval<date> when = (*super_).first;
    // Who is member of group med_users AND admin_users ?
    MemberSetT who = (*super_++).second;
    cout << "[" << first(when) << " - " << last(when) << "]"
         << ": " << who << endl;
}
```

```
}


int main()
{
    cout << ">>Interval Container Library: Sample user_groups.cpp <<\n";
    cout << "-------------------------------------------------------\n";
    user_groups();
    return 0;
}

// Program output:
/*-----------------------------------------------------------------------------
>>Interval Container Library: Sample user_groups.cpp <<
-------------------------------------------------------
----- Membership of medical staff -----------------------------------
[2008-Jan-01 - 2008-Jan-14]: Harry Mary
[2008-Jan-15 - 2008-Jan-31]: Dr.Jekyll Harry Mary
[2008-Feb-01 - 2008-Oct-15]: Dr.Jekyll Harry Mary Mr.Hyde
[2008-Oct-16 - 2008-Dec-31]: Dr.Jekyll Harry Mary
----- Membership of admin staff -------------------------------------
[2008-Jan-15 - 2008-Jan-31]: Dr.Jekyll
[2008-Feb-01 - 2008-Mar-19]: Dr.Jekyll Mr.Hyde
[2008-Mar-20 - 2008-Sep-30]: Diana Dr.Jekyll Mr.Hyde Susan
[2008-Oct-01 - 2008-Oct-15]: Dr.Jekyll Mr.Hyde
[2008-Oct-16 - 2008-Dec-31]: Dr.Jekyll
----- Membership of all users (med + admin) -------------------------
[2008-Jan-01 - 2008-Jan-14]: Harry Mary
[2008-Jan-15 - 2008-Jan-31]: Dr.Jekyll Harry Mary
[2008-Feb-01 - 2008-Mar-19]: Dr.Jekyll Harry Mary Mr.Hyde
[2008-Mar-20 - 2008-Sep-30]: Diana Dr.Jekyll Harry Mary Mr.Hyde Susan
[2008-Oct-01 - 2008-Oct-15]: Dr.Jekyll Harry Mary Mr.Hyde
[2008-Oct-16 - 2008-Dec-31]: Dr.Jekyll Harry Mary
----- Membership of super users: intersection(med,admin) ------------
[2008-Jan-15 - 2008-Jan-31]: Dr.Jekyll
[2008-Feb-01 - 2008-Oct-15]: Dr.Jekyll Mr.Hyde
[2008-Oct-16 - 2008-Dec-31]: Dr.Jekyll
-----------------------------------------------------------------------------*/
```

Std copy

The standard algorithm `std::copy` can be used to fill interval containers from standard containers of intervals or interval value pairs (segments). Because intervals do not represent ***elements*** but ***sets***, that can be empty or contain more than one element, the usage of `std::copy` differs from what we are familiar with using *containers of elements*.

- Use `icl::inserter` from `#include <boost/icl/iterator.hpp>` instead of `std::inserter` to call insertions on the target interval container.

- As shown in the examples above and below this point, most of the time we will not be interested to `insert` segments into `interval_maps` but to ***add*** them, in order to generate the desired aggregation results. You can use `std::copy` with an `icl::adder` instead of an `icl::inserter` to achieve this.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <boost/icl/interval_map.hpp>

using namespace std;
using namespace boost;
using namespace boost::icl;

// 'make_segments' returns a vector of interval value pairs, which
// are not sorted. The values are taken from the minimal example
// in section 'interval combining styles'.
vector<pair<discrete_interval<int>, int> > make_segments()
{
    vector<pair<discrete_interval<int>, int> > segment_vec;
    segment_vec.push_back(make_pair(discrete_interval<int>::right_open(2,4), 1));
    segment_vec.push_back(make_pair(discrete_interval<int>::right_open(4,5), 1));
    segment_vec.push_back(make_pair(discrete_interval<int>::right_open(1,3), 1));
    return segment_vec;
}

// 'show_segments' displays the source segements.
void show_segments(const vector<pair<discrete_interval<int>, int> >& segments)
{
    vector<pair<discrete_interval<int>, int> >::const_iterator iter = segments.begin();
    while(iter != segments.end())
    {
        cout << "(" << iter->first << "," << iter->second << ")";
        ++iter;
    }
}

void std_copy()
{
    // So we have some segments stored in an std container.
    vector<pair<discrete_interval<int>, int> > segments = make_segments();
    // Display the input
    cout << "input sequence: "; show_segments(segments); cout << "\n\n";

    // We are going to 'std::copy' those segments into an interval_map:
    interval_map<int,int> segmap;

    // Use an 'icl::inserter' from <boost/icl/iterator.hpp> to call
    // insertion on the interval container.
    std::copy(segments.begin(), segments.end(),
              icl::inserter(segmap, segmap.end()));
    cout << "icl::inserting: " << segmap << endl;
    segmap.clear();

    // When we are feeding data into interval_maps, most of the time we are
    // intending to compute an aggregation result. So we are not interested
    // the std::insert semantincs but the aggregating icl::addition semantics.
    // To achieve this there is an icl::add_iterator and an icl::adder function
    // provided in <boost/icl/iterator.hpp>.
    std::copy(segments.begin(), segments.end(),
              icl::adder(segmap, segmap.end())); //Aggregating associated values
    cout << "icl::adding   : " << segmap << endl;

    // In this last case, the semantics of 'std::copy' transforms to the
    // generalized addition operation, that is implemented by operator
    // += or + on itl maps and sets.
}
```

```
int main()
{
    cout << ">>    Interval Container Library: Example std_copy.cpp    <<\n";
    cout << "-----------------------------------------------------------\n";
    cout << "Using std::copy to fill an interval_map:\n\n";

    std_copy();
    return 0;
}

// Program output:
/*---------------------------------------------------------
>>    Interval Container Library: Example std_copy.cpp    <<
-----------------------------------------------------------
Using std::copy to fill an interval_map:

input sequence: ([2,4),1)([4,5),1)([1,3),1)

icl::inserting: {([1,5)->1)}
icl::adding   : {([1,2)->1)([2,3)->2)([3,5)->1)}
---------------------------------------------------------*/
```

Std transform

Instead of writing loops, the standard algorithm `std::transform` can be used to fill interval containers from std containers of user defined objects. We need a function, that maps the *user defined object* into the *segement type* of an interval map or the *interval type* of an interval set. Based on that we can use `std::transform` with an `icl::inserter` or `icl::adder` to transform the user objects into interval containers.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <boost/icl/split_interval_map.hpp>
#include <boost/icl/separate_interval_set.hpp>

using namespace std;
using namespace boost;
using namespace boost::icl;

// Suppose we are working with a class called MyObject, containing some
// information about interval bounds e.g. _from, _to and some data members
// that carry associated information like e.g. _value.
class MyObject
{
public:
    MyObject(){}
    MyObject(int from, int to, int value): _from(from), _to(to), _value(value){}
    int from()const {return _from;}
    int to()const   {return _to;}
    int value()const{return _value;}
private:
    int _from;
    int _to;
    int _value;
};

// ... in order to use the std::transform algorithm to fill
// interval maps with MyObject data we need a function
// 'to_segment' that maps an object of type MyObject into
// the value type to the interval map we want to tranform to ...
pair<discrete_interval<int>, int> to_segment(const MyObject& myObj)
{
    return std::pair< discrete_interval<int>, int >
        (discrete_interval<int>::closed(myObj.from(), myObj.to()), myObj.value());
}

// ... there may be another function that returns the interval
// of an object only
discrete_interval<int> to_interval(const MyObject& myObj)
{
    return discrete_interval<int>::closed(myObj.from(), myObj.to());
}


// ... make_object computes a sequence of objects to test.
vector<MyObject> make_objects()
{
    vector<MyObject> object_vec;
    object_vec.push_back(MyObject(2,3,1));
    object_vec.push_back(MyObject(4,4,1));
    object_vec.push_back(MyObject(1,2,1));
    return object_vec;
}

// ... show_objects displays the sequence of input objects.
void show_objects(const vector<MyObject>& objects)
{
    vector<MyObject>::const_iterator iter = objects.begin();
    while(iter != objects.end())
    {
        cout << "([" << iter->from() << "," << iter->to() << "],"
             << iter->value() << ")";
```

```
        ++iter;
    }
}


void std_transform()
{
    // This time we want to transform objects into a splitting interval map:
    split_interval_map<int,int> segmap;
    vector<MyObject> myObjects = make_objects();

    // Display the input
    cout << "input sequence: "; show_objects(myObjects); cout << "\n\n";

    // Use an icl::inserter to fill the interval map via inserts
    std::transform(myObjects.begin(), myObjects.end(),
                   icl::inserter(segmap, segmap.end()),
                   to_segment);
    cout << "icl::inserting: " << segmap << endl;
    segmap.clear();

    // In order to compute aggregation results on associated values, we
    // usually want to use an icl::adder instead of an std or icl::inserter
    std::transform(myObjects.begin(), myObjects.end(),
                   icl::adder(segmap, segmap.end()),
                   to_segment);
    cout << "icl::adding   : " << segmap << "\n\n";

    separate_interval_set<int> segset;
    std::transform(myObjects.begin(), myObjects.end(),
                   icl::adder   (segset, segset.end()),
    // could be a  icl::inserter(segset, segset.end()), here: same effect
                   to_interval);

    cout << "Using std::transform to fill a separate_interval_set:\n\n";
    cout << "icl::adding   : " << segset << "\n\n";
}


int main()
{
    cout << ">>  Interval Container Library: Example std_transform.cpp  <<\n";
    cout << "------------------------------------------------------------\n";
    cout << "Using std::transform to fill a split_interval_map:\n\n";

    std_transform();
    return 0;
}

// Program output:
/*------------------------------------------------------------
>>  Interval Container Library: Example std_transform.cpp  <<
------------------------------------------------------------
Using std::transform to fill a split_interval_map:

input sequence: ([2,3],1)([4,4],1)([1,2],1)

icl::inserting: {([1,2)->1)([2,3]->1)([4,4]->1)}
```

```
icl::adding   : {([1,2)->1)([2,2]->2)((2,3]->1)([4,4]->1)}

Using std::transform to fill a separate_interval_set:

icl::adding   : {[1,3][4,4]}
-----------------------------------------------------------*/
```

To get clear about the different behaviors of interval containers in the example, you may want to refer to the section about interval combining styles that uses the same data.

Custom interval

Example **custom interval** demonstrates how to use interval containers with an own interval class type.

```
#include <iostream>
#include <boost/icl/interval_set.hpp>

using namespace std;
using namespace boost::icl;

// Here is a typical class that may model intervals in your application.
class MyInterval
{
public:
    MyInterval(): _first(), _past(){}
    MyInterval(int lo, int up): _first(lo), _past(up){}
    int first()const{ return _first; }
    int past ()const{ return _past; }
private:
    int _first, _past;
};

namespace boost{ namespace icl
{
// Class template interval_traits serves as adapter to register and customize your interval class
template<>
struct interval_traits< MyInterval >       //1.  Partially specialize interval_traits for
{                                          //    your class MyInterval
                                           //2.  Define associated types
    typedef MyInterval     interval_type;  //2.1 MyInterval will be the interval_type
    typedef int            domain_type;    //2.2 The elements of the domain are ints
    typedef std::less<int> domain_compare; //2.3 This is the way our element shall be ordered.
                                           //3.  Next we define the essential functions
                                           //    of the specialisation
                                           //3.1 Construction of intervals
    static interval_type construct(const domain_type& lo, const domain_type& up)
    { return interval_type(lo, up); }
                                           //3.2 Selection of values
    static domain_type lower(const interval_type& inter_val){ return inter_val.first(); };
    static domain_type upper(const interval_type& inter_val){ return inter_val.past(); };
};

template<>
struct interval_bound_type<MyInterval>     //4.  Finally we define the interval borders.
{                                          //    Choose between static_open         (lo..up)
    typedef interval_bound_type type;      //                   static_left_open    (lo..up]
    BOOST_STATIC_CONSTANT(bound_type, value = interval_bounds::static_right_open);
};                                         //               and static_closed       [lo..up]

}} // namespace boost icl
```

```
void custom_interval()
{
    // Now we can use class MyInterval with interval containers:
    typedef interval_set<int, std::less, MyInterval> MyIntervalSet;
    MyIntervalSet mySet;
    mySet += MyInterval(1,9);
    cout << mySet << endl;
    mySet.subtract(3) -= 6;
    cout << mySet << "            subtracted 3 and 6\n";
    mySet ^= MyInterval(2,8);
    cout << mySet <<      "  flipped between 2 and 7\n";
}


int main()
{
    cout << ">>Interval Container Library: Sample custom_interval.cpp <<\n";
    cout << "-----------------------------------------------------------\n";
    cout << "This program uses a user defined interval class:\n";
    custom_interval();
    return 0;
}

// Program output:
/*-----------------------------------------------------------------------------
>>Interval Container Library: Sample custom_interval.cpp <<
-----------------------------------------------------------
This program uses a user defined interval class:
{[1,                      9)}
{[1,  3)   [4,  6)   [7,  9)}       subtracted 3 and 6
{[1,2) [3,4)     [6,7) [8,9)}  flipped between 2 and 7
-----------------------------------------------------------------------------*/
```

Projects

Projects are examples on the usage of interval containers that go beyond small toy snippets of code. The code presented here addresses more serious applications that approach the quality of real world programming. At the same time it aims to guide the reader more deeply into various aspects of the library. In order not to overburden the reader with implementation details, the code in ***projects*** tries to be ***minimal***. It has a focus on the main aspects of the projects and is not intended to be complete and mature like the library code itself. Cause it's minimal, project code lives in `namespace mini`.

Large Bitset

Bitsets are just sets. Sets of unsigned integrals, to be more precise. The prefix ***bit*** usually only indicates, that the representation of those sets is organized in a compressed form that exploits the fact, that we can switch on an off single bits in machine words. Bitsets are therefore known to be very small and thus efficient. The efficiency of bitsets is usually coupled to the precondition that the range of values of elements is relatively small, like [0..32) or [0..64), values that can be typically represented in single or a small number of machine words. If we wanted to represent a set containing two values {1, 1000000}, we would be much better off using other sets like e.g. an `std::set`.

Bitsets compress well, if elements spread over narrow ranges only. Interval sets compress well, if many elements are clustered over intervals. They can span large sets very efficiently then. In project ***Large Bitset*** we want to ***combine the bit compression and the interval compression*** to achieve a set implementation, that is capable of spanning large chunks of contiguous elements using intervals and also to represent more narrow *nests* of varying bit sequences using bitset compression. As we will see, this can be achieved using only a small amount of code because most of the properties we need are provided by an `interval_map` of `bitsets`:

```
typedef interval_map<IntegralT, SomeBitSet<N>, partial_absorber,
                     std::less, inplace_bit_add, inplace_bit_and> IntervalBitmap;
```

Such an `IntervalBitmap` represents `k*N` bits for every segment.

```
[a, a+k)->'1111....1111' // N bits associated: Represents a total of k*N bits.
```

For the interval `[a, a+k)` above all bits are set. But we can also have individual *nests* or *clusters* of bitsequences.

```
[b,  b+1)->'01001011...1'
[b+1,b+2)->'11010001...0'
. . .
```

and we can span intervals of equal bit sequences that represent periodic patterns.

```
[c,d)->'010101....01'  // Every second bit is set              in range [c,d)
[d,e)->'001100..0011'  // Every two bits alternate             in range [d,e)
[e,f)->'bit-sequence'  // 'bit-sequence' reoccurs every N bits in range [e,f)
```

An `IntervalBitmap` can represent `N*(2^M)` elements, if `M` is the number of bits of the integral type `IntegralT`. Unlike bitsets, that usually represent ***unsigned*** integral numbers, large_bitset may range over negative numbers as well. There are fields where such large bitsets implementations are needed. E.g. for the compact representation of large file allocation tables. What remains to be done for project **Large Bitset** is to code a wrapper `class large_bitset` around `IntervalBitmap` so that `large_bitset` looks and feels like a usual set class.

Using large_bitset

To quicken your appetite for a look at the implementation here are a few use cases first. Within the examples that follow, we will use `nat`*k* for unsigned integrals and `bits`*k* for bitsets containing *k* bits.

Let's start large. In the first example . . .

```
void test_large()
{
    const nat64 much = 0xffffffffffffffffull;
    large_bitset<> venti; // ... the largest, I can think of ;)
    venti += discrete_interval<nat64>(0, much);

    cout << "----- Test function test_large() -----------------------------------------------
\n";
   cout << "We have just turned on the awesome amount of 18,446,744,073,709,551,616 bits ;-)\n";
    venti.show_segments();
```

. . . we are testing the limits. First we set all bits and then we switch off the very last bit.

```
    cout << "---- Let's swich off the very last bit -----------------------------------------
\n";
    venti -= much;
    venti.show_segments();

    cout << "---- Venti is plenty ... let's do something small: A tall ----------------------
\n\n";
}
```

Program output (*a little beautified*):

```
----- Test function test_large() -----------------------------------------------
We have just turned on the awesome amount of 18,446,744,073,709,551,616 bits ;-)
[                ↵
 0, 288230376151711744) -> 1111111111111111111111111111111111111111111111111111111111111111
---- Let's swich off the very last bit -----------------------------------------
[                 0, 288230376151711743) -> ↵
1111111111111111111111111111111111111111111111111111111111111111
[288230376151711743, 288230376151711744) -> ↵
1111111111111111111111111111111111111111111111111111111111111110
---- Venti is plenty ... let's do something small: A tall ----------------------
```

More readable is a smaller version of `large_bitset`. In function `test_small()` we apply a few more operations . . .

```
void test_small()
{
    large_bitset<nat32, bits8> tall; // small is tall ...
        // ... because even this 'small' large_bitset
        // can represent up to 2^32 == 4,294,967,296 bits.

    cout << "----- Test function test_small() -----------\n";
    cout << "-- Switch on all bits in range [0,64] ------\n";
    tall += discrete_interval<nat>(0, 64);
    tall.show_segments();
    cout << "--------------------------------------------\n";

    cout << "-- Turn off bits: 25,27,28 -----------------\n";

    (((tall -= 25) -= 27) -= 28) ;
    tall.show_segments();
    cout << "--------------------------------------------\n";

    cout << "-- Flip bits in range [24,30) --------------\n";
    tall ^= discrete_interval<nat>::right_open(24,30);
    tall.show_segments();
    cout << "--------------------------------------------\n";

    cout << "-- Remove the first 10 bits ----------------\n";
    tall -= discrete_interval<nat>::right_open(0,10);
    tall.show_segments();

    cout << "-- Remove even bits in range [0,72) --------\n";
    int bit;
    for(bit=0; bit<72; bit++) if(!(bit%2)) tall -= bit;
    tall.show_segments();

    cout << "--    Set odd  bits in range [0,72) --------\n";
    for(bit=0; bit<72; bit++) if(bit%2) tall += bit;
    tall.show_segments();

    cout << "--------------------------------------------\n\n";

}
```

. . . producing this output:

```
----- Test function test_small() -----------
-- Switch on all bits in range [0,64] ------
[0,8)->11111111
[8,9)->10000000
--------------------------------------------
-- Turn off bits: 25,27,28 -----------------
[0,3)->11111111
[3,4)->10100111
[4,8)->11111111
[8,9)->10000000
--------------------------------------------
-- Flip bits in range [24,30) --------------
[0,3)->11111111
[3,4)->01011011
[4,8)->11111111
[8,9)->10000000
--------------------------------------------
-- Remove the first 10 bits ----------------
[1,2)->00111111
[2,3)->11111111
[3,4)->01011011
[4,8)->11111111
[8,9)->10000000
-- Remove even bits in range [0,72) --------
[1,2)->00010101
[2,3)->01010101
[3,4)->01010001
[4,8)->01010101
--    Set odd  bits in range [0,72) --------
[0,9)->01010101
--------------------------------------------
```

Finally, we present a little *picturesque* example, that demonstrates that `large_bitset` can also serve as a self compressing bitmap, that we can 'paint' with.

```
void test_picturesque()
{
    typedef large_bitset<nat, bits8> Bit8Set;

    Bit8Set square, stare;
    square += discrete_interval<nat>(0,8);
    for(int i=1; i<5; i++)
    {
        square += 8*i;
        square += 8*i+7;
    }

    square += discrete_interval<nat>(41,47);

    cout << "----- Test function test_picturesque() -----\n";
    cout << "-------- empty face:       "
         << square.interval_count()           << " intervals -----\n";
    square.show_matrix(" *");

    stare += 18; stare += 21;
    stare += discrete_interval<nat>(34,38);

    cout << "-------- compressed smile: "
         << stare.interval_count()            << " intervals -----\n";
    stare.show_matrix(" *");

    cout << "-------- staring bitset:   "
         << (square + stare).interval_count() << " intervals -----\n";
    (square + stare).show_matrix(" *");

    cout << "--------------------------------------------\n";
}
```

Note that we have two `large_bitsets` for the *outline* and the *interior*. Both parts are compressed but we can compose both by `operator +`, because the right *positions* are provided. This is the program output:

```
----- Test function test_picturesque() -----
-------- empty face:       3 intervals -----
********
*      *
*      *
*      *
*      *
 ******
-------- compressed smile: 2 intervals -----
  *  *
  ****
-------- staring bitset:   6 intervals -----
********
*      *
* *  * *
*      *
* **** *
 ******
--------------------------------------------
```

So, may be you are curious how this class template is coded on top of `interval_map` using only about 250 lines of code. This is shown in the sections that follow.

The interval_bitmap

To begin, let's look at the basic data type again, that will be providing the major functionality:

```
typedef interval_map<DomainT, BitSetT, partial_absorber,
                     std::less, inplace_bit_add, inplace_bit_and> IntervalBitmap;
```

`DomainT` is supposed to be an integral type, the bitset type `BitSetT` will be a wrapper class around an unsigned integral type. `BitSetT` has to implement bitwise operators that will be called by the functors `inplace_bit_add<BitSetT>` and `inplace_bit_and<BitSetT>`. The type trait of interval_map is `partial_absorber`, which means that it is *partial* and that empty `BitSetT`s are not stored in the map. This is desired and keeps the `interval_map` minimal, storing only bitsets, that contain at least one bit switched on. Functor template `inplace_bit_add` for parameter `Combine` indicates that we do not expect `operator +=` as addition but the bitwise operator `|=`. For template parameter `Section` which is instaniated by `inplace_bit_and` we expect the bitwise `&=` operator.

A class implementation for the bitset type

The code of the project is enclosed in a `namespace mini`. The name indicates, that the implementation is a *minimal* example implementation. The name of the bitset class will be `bits` or `mini::bits` if qualified.

To be used as a codomain parameter of class template `interval_map`, `mini::bits` has to implement all the functions that are required for a codomain_type in general, which are the default constructor `bits()` and an equality `operator==`. Moreover `mini::bits` has to implement operators required by the instantiations for parameter `Combine` and `Section` which are `inplace_bit_add` and `inplace_bit_and`. From functors `inplace_bit_add` and `inplace_bit_and` there are inverse functors `inplace_bit_subtract` and `inplace_bit_xor`. Those functors use operators `|= &= ^=` and `~`. Finally if we want to apply lexicographical and subset comparison on large_bitset, we also need an `operator <`. All the operators that we need can be implemented for `mini::bits` on a few lines:

```
template<class NaturalT> class bits
{
public:
    typedef NaturalT word_type;
    static const int       digits = std::numeric_limits<NaturalT>::digits;
    static const word_type w1     = static_cast<NaturalT>(1) ;

    bits():_bits(){}
    explicit bits(word_type value):_bits(value){}

    word_type word()const{ return _bits; }
    bits& operator |= (const bits& value){_bits |= value._bits; return *this;}
    bits& operator &= (const bits& value){_bits &= value._bits; return *this;}
    bits& operator ^= (const bits& value){_bits ^= value._bits; return *this;}
    bits  operator ~  ()const { return bits(~_bits); }
    bool operator  <  (const bits& value)const{return _bits < value._bits;}
    bool operator  == (const bits& value)const{return _bits == value._bits;}

    bool contains(word_type element)const{ return ((w1 << element) & _bits) != 0; }
    std::string as_string(const char off_on[2] = " 1")const;

private:
    word_type _bits;
};
```

Finally there is one important piece of meta information, we have to provide: `mini::bits` has to be recognized as a `Set` by the icl code. Otherwise we can not exploit the fact that a map of sets is model of `Set` and the resulting large_bitset would not behave like a set. So we have to say that `mini::bits` shall be sets:

```
namespace boost { namespace icl
{
    template<class NaturalT>
    struct is_set<mini::bits<NaturalT> >
    {
        typedef is_set<mini::bits<NaturalT> > type;
        BOOST_STATIC_CONSTANT(bool, value = true);
    };

    template<class NaturalT>
    struct has_set_semantics<mini::bits<NaturalT> >
    {
        typedef has_set_semantics<mini::bits<NaturalT> > type;
        BOOST_STATIC_CONSTANT(bool, value = true);
    };
}}
```

This is done by adding a partial template specialization to the type trait template `icl::is_set`. For the extension of this type trait template and the result values of inclusion_compare we need these `#includes` for the implementation of `mini::bits`:

```
                                               // These includes are needed ...
#include <string>                              // for conversion to output and to
#include <boost/icl/type_traits/has_set_semantics.hpp>//declare that bits has the
                                               // behavior of a set.
```

Implementation of a large bitset

Having finished our `mini::bits` implementation, we can start to code the wrapper class that hides the efficient interval map of mini::bits and exposes a simple and convenient set behavior to the world of users.

Let's start with the required `#includes` this time:

```
#include <iostream>                   // to organize output
#include <limits>                     // limits and associated constants
#include <boost/operators.hpp>        // to define operators with minimal effort
#include "meta_log.hpp"               // a meta logarithm
#include "bits.hpp"                   // a minimal bitset implementation
#include <boost/icl/interval_map.hpp> // base of large bitsets

namespace mini // minimal implementations for example projects
{
```

Besides `boost/icl/interval_map.hpp` and `bits.hpp` the most important include here is `boost/operators.hpp`. We use this library in order to further minimize the code and to provide pretty extensive operator functionality using very little code.

For a short and concise naming of the most important unsigned integer types and the corresponding `mini::bits` we define this:

```
typedef unsigned char      nat8; // nati i: number bits
typedef unsigned short     nat16;
typedef unsigned long      nat32;
typedef unsigned long long nat64;
typedef unsigned long      nat;

typedef bits<nat8>  bits8;
typedef bits<nat16> bits16;
typedef bits<nat32> bits32;
typedef bits<nat64> bits64;
```

large_bitset: Public interface

And now let's code `large_bitset`.

```
template
<
    typename    DomainT = nat64,
    typename    BitSetT = bits64,
    ICL_COMPARE Compare = ICL_COMPARE_INSTANCE(std::less, DomainT),
   ICL_INTERVAL(ICL_COMPARE) Interval = ICL_INTERVAL_INSTANCE(ICL_INTERVAL_DEFAULT, DomainT, Com↵
pare),
    ICL_ALLOC   Alloc   = std::allocator
>
class large_bitset
    : boost::equality_comparable < large_bitset<DomainT,BitSetT,Compare,Interval,Alloc>
    , boost::less_than_comparable< large_bitset<DomainT,BitSetT,Compare,Interval,Alloc>

    , boost::addable       < large_bitset<DomainT,BitSetT,Compare,Interval,Alloc>
    , boost::orable        < large_bitset<DomainT,BitSetT,Compare,Interval,Alloc>
    , boost::subtractable  < large_bitset<DomainT,BitSetT,Compare,Interval,Alloc>
    , boost::andable       < large_bitset<DomainT,BitSetT,Compare,Interval,Alloc>
    , boost::xorable       < large_bitset<DomainT,BitSetT,Compare,Interval,Alloc>

    , boost::addable2      < large_bitset<DomainT,BitSetT,Compare,Interval,Alloc>, DomainT
    , boost::orable2       < large_bitset<DomainT,BitSetT,Compare,Interval,Alloc>, DomainT
    , boost::subtractable2 < large_bitset<DomainT,BitSetT,Compare,Interval,Alloc>, DomainT
    , boost::andable2      < large_bitset<DomainT,BitSetT,Compare,Interval,Alloc>, DomainT
    , boost::xorable2      < large_bitset<DomainT,BitSetT,Compare,Interval,Alloc>, DomainT

    , boost::addable2      < large_bitset<DomainT,BitSetT,Compare,Interval,Alloc>, ICL_INTER↵
VAL_TYPE(Interval,DomainT,Compare)
    , boost::orable2       < large_bitset<DomainT,BitSetT,Compare,Interval,Alloc>, ICL_INTER↵
VAL_TYPE(Interval,DomainT,Compare)
    , boost::subtractable2 < large_bitset<DomainT,BitSetT,Compare,Interval,Alloc>, ICL_INTER↵
VAL_TYPE(Interval,DomainT,Compare)
    , boost::andable2      < large_bitset<DomainT,BitSetT,Compare,Interval,Alloc>, ICL_INTER↵
VAL_TYPE(Interval,DomainT,Compare)
    , boost::xorable2      < large_bitset<DomainT,BitSetT,Compare,Interval,Alloc>, ICL_INTER↵
VAL_TYPE(Interval,DomainT,Compare)
      > > > > > > > > > > > > > > > > >
    //^ & - | + ^ & - | + ^ & - | + < ==
    //segment   element   container
```

The first template parameter `DomainT` will be instantiated with an integral type that defines the kind of numbers that can be elements of the set. Since we want to go for a large set we use `nat64` as default which is a 64 bit unsigned integer ranging from `0` to `2^64-1`. As bitset parameter we also choose a 64-bit default. Parameters `Combine` and `Interval` are necessary to be passed to dependent type expressions. An allocator can be chosen, if desired.

The nested list of private inheritance contains groups of template instantiations from Boost.Operator, that provides derivable operators from more fundamental once. Implementing the fundamental operators, we get the derivable ones *for free*. Below is a short overview of what we get using Boost.Operator, where **S** stands for `large_bitset`, **i** for it's `interval_type` and **e** for it's `domain_type` or `element_type`.

Group	fundamental	derivable
Equality, ordering	`==`	`!=`
	`<`	`> <= >=`
Set operators (**S** x **S**)	`+= |= -= &= ^=`	`+ | - & ^`
Set operators (**S** x **e**)	`+= |= -= &= ^=`	`+ | - & ^`
Set operators (**S** x **i**)	`+= |= -= &= ^=`	`+ | - & ^`

There is a number of associated types

```
typedef boost::icl::interval_map
    <DomainT, BitSetT, boost::icl::partial_absorber,
     std::less, boost::icl::inplace_bit_add, boost::icl::inplace_bit_and> interval_bitmap_type;

typedef DomainT                                     domain_type;
typedef DomainT                                     element_type;
typedef BitSetT                                     bitset_type;
typedef typename BitSetT::word_type                 word_type;
typedef typename interval_bitmap_type::interval_type interval_type;
typedef typename interval_bitmap_type::value_type    value_type;
```

most importantly the implementing `interval_bitmap_type` that is used for the implementing container.

```
private:
    interval_bitmap_type _map;
```

In order to use Boost.Operator we have to implement the fundamental operators as class members. This can be done quite schematically.

```
public:
    bool     operator ==(const large_bitset& rhs)const { return _map == rhs._map; }
    bool     operator < (const large_bitset& rhs)const { return _map <  rhs._map; }

    large_bitset& operator +=(const large_bitset& rhs) {_map += rhs._map; return *this;}
    large_bitset& operator |=(const large_bitset& rhs) {_map |= rhs._map; return *this;}
    large_bitset& operator -=(const large_bitset& rhs) {_map -= rhs._map; return *this;}
    large_bitset& operator &=(const large_bitset& rhs) {_map &= rhs._map; return *this;}
    large_bitset& operator ^=(const large_bitset& rhs) {_map ^= rhs._map; return *this;}

    large_bitset& operator +=(const element_type& rhs) {return add(interval_type(rhs));       }
    large_bitset& operator |=(const element_type& rhs) {return add(interval_type(rhs));       }
    large_bitset& operator -=(const element_type& rhs) {return subtract(interval_type(rhs)); }
    large_bitset& operator &=(const element_type& rhs) {return intersect(interval_type(rhs));}
    large_bitset& operator ^=(const element_type& rhs) {return flip(interval_type(rhs));      }

    large_bitset& operator +=(const interval_type& rhs){return add(rhs);       }
    large_bitset& operator |=(const interval_type& rhs){return add(rhs);       }
    large_bitset& operator -=(const interval_type& rhs){return subtract(rhs); }
    large_bitset& operator &=(const interval_type& rhs){return intersect(rhs);}
    large_bitset& operator ^=(const interval_type& rhs){return flip(rhs);      }
```

As we can see, the seven most important operators that work on the class type `large_bitset` can be directly implemented by propagating the operation to the implementing `_map` of type `interval_bitmap_type`. For the operators that work on segment and element types, we use member functions `add`, `subtract`, `intersect` and `flip`. As we will see only a small amount of adaper code is needed to couple those functions with the functionality of the implementing container.

Member functions `add`, `subtract`, `intersect` and `flip`, that allow to combine ***intervals*** to `large_bitsets` can be uniformly implemented using a private function `segment_apply` that applies *addition*, *subtraction*, *intersection* or *symmetric difference*, after having translated the interval's borders into the right bitset positions.

```
large_bitset& add      (const interval_type& rhs){return segment_apply(&large_bitset::add_,   ↵
  rhs);}
large_bitset& subtract (const interval_type& rhs){return segment_apply(&large_bitset::sub↵
tract_, rhs);}
large_bitset& intersect(const interval_type& rhs){return segment_apply(&large_bitset::inter↵
sect_,rhs);}
large_bitset& flip     (const interval_type& rhs){return segment_apply(&large_bitset::flip_,  ↵
  rhs);}
```

In the sample programs, that we will present to demonstrate the capabilities of `large_bitset` we will need a few additional functions specifically output functions in two different flavors.

```
size_t interval_count()const { return boost::icl::interval_count(_map); }

void show_segments()const
{
    for(typename interval_bitmap_type::const_iterator it_ = _map.begin();
        it_ != _map.end(); ++it_)
    {
        interval_type   itv  = it_->first;
        bitset_type     bits = it_->second;
        std::cout << itv << "->" << bits.as_string("01") << std::endl;
    }
}

void show_matrix(const char off_on[2] = " 1")const
{
    using namespace boost;
    typename interval_bitmap_type::const_iterator iter = _map.begin();
    while(iter != _map.end())
    {
        element_type fst = icl::first(iter->first), lst = icl::last(iter->first);
        for(element_type chunk = fst; chunk <= lst; chunk++)
            std::cout << iter->second.as_string(off_on) << std::endl;
        ++iter;
    }
}
```

- The first one, `show_segments()` shows the container content as it is implemented, in the compressed form.
- The second function `show_matrix` shows the complete matrix of bits that is represented by the container.

large_bitset: Private implementation

In order to implement operations like the addition of an element say `42` to the large bitset, we need to translate the *value* to the *position* of the associated **bit** representing `42` in the interval container of bitsets. As an example, suppose we use a

```
large_bitset<nat, mini::bits8> lbs;
```

that carries small bitsets of 8 bits only. The first four interval of `lbs` are assumed to be associated with some bitsets. Now we want to add the interval `[a,b]==[5,27]`. This will result in the following situation:

```
   [0,1)->   [1,2)->   [2,3)->   [3,4)->
   [00101100][11001011][11101001][11100000]
+       [111  11111111  11111111  1111]      [5,27] as bitset
         a                           b

=> [0,1)->   [1,3)->   [3,4)->
   [00101111][11111111][11110000]
```

So we have to convert values 5 and 27 into a part that points to the interval and a part that refers to the position within the interval, which is done by a *division* and a *modulo* computation. (In order to have a consistent representation of the bitsequences across the containers, within this project, bitsets are denoted with the ***least significant bit on the left!***)

```
A = a/8 =  5/8 = 0 // refers to interval
B = b/8 = 27/8 = 3
R = a%8 =  5%8 = 5 // refers to the position in the associated bitset.
S = b%8 = 27%8 = 3
```

All *division* and *modulo operations* needed here are always done using a divisor `d` that is a power of `2`: `d = 2^x`. Therefore division and modulo can be expressed by bitset operations. The constants needed for those bitset computations are defined here:

```
private:                                              // Example value
    static const word_type                            //   8-bit case
        digits  = std::numeric_limits                // ----------------------------------------------
----------------
                    <word_type>::digits       , //   8          Size of the associated bitsets
       divisor = digits                       , //   8        Divisor to find intervals for values
        last    = digits-1                    , //   7            Last bit (0 based)
       shift   = log2_<divisor>::value       , //   3          To express the division as bit shift
        w1      = static_cast<word_type>(1) , //                 Helps to avoid static_casts for ↵
long long
        mask    = divisor - w1               , //   7=11100000  Helps to express the modulo oper↵
ation as bit_and
        all     = ~static_cast<word_type>(0), // 255=11111111  Helps to express a complete asso↵
ciated bitset
        top     = w1 << (digits-w1)      ;     // 128=00000001  Value of the most significant ↵
bit of associated bitsets
                                              //             !> Note: Most signigicant bit on ↵
the right.
```

Looking at the example again, we can see that we have to identify the positions of the beginning and ending of the interval [5,27] that is to insert, and then ***subdivide*** that range of bitsets into three partitions.

1. The bitset where the interval starts.
2. the bitset where the interval ends
3. The bitsets that are completely overlapped by the interval

```
combine interval [5,27] to large_bitset lbs w.r.t. some operation o

   [0,1)->   [1,2)->   [2,3)->   [3,4)->
   [00101100][11001011][11101001][11100000]
o       [111  11111111  11111111  1111]
         a                           b
subdivide:
   [first!  ][mid_1] . . .[mid_n][   !last]
   [00000111][1...1] . . .[1...1][11110000]
```

After subdividing, we perform the operation `o` as follows:

1. For the first bitset: Set all bits from ther starting bit (`!`) to the end of the bitset to `1`. All other bits are `0`. Then perform operation `o`: `_map o= ([0,1)->00000111)`

2. For the last bitset: Set all bits from the beginning of the bitset to the ending bit (`!`) to `1`. All other bits are `0`. Then perform operation `o`: `_map o= ([3,4)->11110000)`

3. For the range of bitsets in between the staring and ending one, perform operation `o`: `_map o= ([1,3)->11111111)`

The algorithm, that has been outlined and illustrated by the example, is implemented by the private member function `segment_apply`. To make the combiner operation a variable in this algorithm, we use a *pointer to member function type*

```
typedef void (large_bitset::*segment_combiner)(element_type, element_type, bitset_type);
```

as first function argument. We will pass member functions `combine_` here,

```
combine_(first_of_interval, end_of_interval, some_bitset);
```

that take the beginning and ending of an interval and a bitset and combine them to the implementing `interval_bitmap_type _map`. Here are these functions:

```
void       add_(DomainT lo, DomainT up, BitSetT bits){_map += value_type(inter↵
val_type::right_open(lo,up), bits);}
void  subtract_(DomainT lo, DomainT up, BitSetT bits){_map -= value_type(inter↵
val_type::right_open(lo,up), bits);}
void intersect_(DomainT lo, DomainT up, BitSetT bits){_map &= value_type(inter↵
val_type::right_open(lo,up), bits);}
void      flip_(DomainT lo, DomainT up, BitSetT bits){_map ^= value_type(inter↵
val_type::right_open(lo,up), bits);}
```

Finally we can code function `segment_apply`, that does the partitioning and subsequent combining:

```
large_bitset& segment_apply(segment_combiner combine, const interval_type& operand)
{
    using namespace boost;
    if(icl::is_empty(operand))
        return *this;
                                                        // same as
    element_type   base = icl::first(operand) >> shift, // icl::first(operand) / divisor
                   ceil = icl::last (operand) >> shift; // icl::last (operand) / divisor
    word_type base_rest = icl::first(operand) &  mask , // icl::first(operand) % divisor
              ceil_rest = icl::last (operand) &  mask ; // icl::last (operand) % divisor

    if(base == ceil) // [first, last] are within one bitset (chunk)
        (this->*combine)(base, base+1, bitset_type(  to_upper_from(base_rest)
                                                   & from_lower_to(ceil_rest)));
    else // [first, last] spread over more than one bitset (chunk)
    {
        element_type mid_low = base_rest == 0   ? base   : base+1, // first element of mid part
                     mid_up  = ceil_rest == all ? ceil+1 : ceil  ; // last  element of mid part

        if(base_rest > 0)    // Bitset of base interval has to be filled from base_rest to last
            (this->*combine)(base, base+1, bitset_type(to_upper_from(base_rest)));
        if(ceil_rest < all)  // Bitset of ceil interval has to be filled from first to ceil_rest
            (this->*combine)(ceil, ceil+1, bitset_type(from_lower_to(ceil_rest)));
        if(mid_low < mid_up) // For the middle part all bits have to set.
            (this->*combine)(mid_low, mid_up, bitset_type(all));
    }
    return *this;
}
```

The functions that help filling bitsets to and from a given bit are implemented here:

```
static word_type from_lower_to(word_type bit){return bit==last ? all : (w1<<(bit+w1))-w1;}
static word_type to_upper_from(word_type bit){return bit==last ? top : ~((w1<<bit)-w1); }
```

This completes the implementation of class template `large_bitset`. Using only a small amount of mostly schematic code, we have been able to provide a pretty powerful, self compressing and generally usable set type for all integral domain types.

Concepts

Naming

The **icl** is about sets and maps and a useful implementation of sets and maps using intervals. In the documentation of the **icl** the different set and map types are grouped in various ways. In order to distinguish those groups we use a naming convention.

Names of concepts start with a capital letter. So `Set` and `Map` stand for the *concept* of a set and a map as defined in the **icl**. When we talk about `Sets` and `Maps` though, most of the time we do not not talk about the concepts themselves but the set of types that implement those concepts in the **icl**. The main groups, ***icl containers*** can be divided in, are summarized in the next table:

	`Set`	`Map`
element container	`std::set`	`icl::map`
interval container	`interval_set`, `separate_interval_set`, `split_interval_set`	`interval_map`, `split_interval_map`

- Containers std:set, `interval_set`, `separate_interval_set`, `split_interval_set` are models of concept `Set`.
- Containers `icl::map`, `interval_map`, `split_interval_map` are models of concept `Map`.
- Containers that are ***implemented*** using elements or element value pairs are called ***element containers***.
- Containers that are ***implemented*** using intervals or interval value pairs (also called segments) are called ***interval containers***.
- When we talk about `Sets` or `Maps` we abstract from the way they are implemented.
- When we talk about *element containers* or *interval containers* we refer to the way they are implemented.
- `std::set` is a model of the icl's `Set` concept.
- `std::map` is ***not*** a model of the icl's `Map` concept.
- The **icl's** element map is always denoted qualified as `icl::map` to avoid confusion with`std::map`.

Aspects

There are two major ***aspects*** or ***views*** of icl containers. The first and predominant aspect is called ***fundamental***. The second and minor aspect is called ***segmental***.

	Fundamental	**Segmental**
Abstraction level	more abstract	less abstract
	sequence of elements is irrelevant	sequence of elements is relevant
	iterator independent	iterator dependent
Informs about	membership of elements	sequence of intervals (segmentation)
Equality	equality of elements	equality of segments
Practical	interval_sets(maps) can be used as sets(maps) of elements(element value pairs)	Segmentation information is available. See e.g. Time grids for months and weeks

On the fundamental aspect

- an `interval` implements a set of elements partially.
- an `interval_set` implements a set of elements.
- an `interval_map` implements a map of element value pairs.

On the segmental aspect

- an `interval_set` implements a set of intervals.
- an `interval_map` implements a map of interval value pairs.

Sets and Maps

A Set Concept

On the fundamental aspect all `interval_sets` are models of a concept `Set`. The `Set` concept of the Interval Template Library refers to the mathematical notion of a set.

Function	Variant	implemented as
empty set		`Set::Set()`
subset relation		`bool Set::within(const Set& s1, const Set& s2)const`
equality		`bool is_element_equal(const Set& s1, const Set& s2)`
set union	inplace	`Set& operator += (Set& s1, const Set& s2)`
		`Set operator + (const Set& s1, const Set& s2)`
set difference	inplace	`Set& operator -= (Set& s1, const Set& s2)`
		`Set operator - (const Set& s1, const Set& s2)`
set intersection	inplace	`Set& operator &= (Set& s1, const Set& s2)`
		`Set operator & (const Set& s1, const Set& s2)`

Equality on `Sets` is not implemented as `operator ==`, because `operator ==` is used for the stronger lexicographical equality on segments, that takes the segmentation of elements into account.

Being models of concept `Set`, `std::set` and all `interval_sets` implement these operations and obey the associated laws on `Sets`. See e.g. an algebra of sets here.

Making intervals complete

An `interval` is considered to be a set of elements as well. With respect to the `Set` concept presented above `interval` implements the concept only partially. The reason for that is that addition and subtraction can not be defined on `intervals`. Two intervals

[1,2] and [4,5] are not addable to a ***single*** new interval. In other words intervals are incomplete w.r.t. union and difference. Interval_sets can be defined as the ***completion*** of intervals for the union and difference operations.

When we claim that addition or subtraction can not be defined on intervals, we are not considering things like e.g. interval arithmetics, where these operations can be defined, but with a different semantics.

A Map Concept

On the fundamental aspect icl::map and all interval_maps are models of a concept Map. Since a map is a set of pairs, we try to design the Map concept in accordance to the Set concept above.

Function	Variant	implemented as
empty map		`Map::Map()`
subset relation		`bool within(const Map& s2, const Map& s2)const`
equality		`bool is_element_equal(const Map& s1, const Map& s2)`
map union	inplace	`Map& operator += (Map& s1, const Map& s2)`
		`Map operator + (const Map& s1, const Map& s2)`
map difference	inplace	`Map& operator -= (Map& s1, const Map& s2)`
		`Map operator - (const Map& s1, const Map& s2)`
map intersection	inplace	`Map& operator &= (Map& s1, const Map& s2)`
		`Map operator & (const Map& s1, const Map& s2)`

As one can see, on the abstract kernel the signatures of the icl's Set and Map concepts are identical, except for the typename. While signatures are identical The sets of valid laws are different, which will be described in more detail in the sections on the semantics of icl Sets and Maps. These semantic differences are mainly based on the implementation of the pivotal member functions add and subtract for elements and intervals that again serve for implementing operator += and operator -=.

Addability, Subtractability and Aggregate on Overlap

While ***addition*** and ***subtraction*** on Sets are implemented as ***set union*** and ***set difference***, for Maps we want to implement ***aggregation*** on the associated values for the case of collision (of key elements) or overlap (of key intervals), which has been refered to as ***aggregate on overlap*** above. This kind of Addability and Subtractability allows to compute a lot of useful aggregation results on an interval_map's associated values, just by adding and subtracting value pairs. Various examples of ***aggregate on overlap*** are given in section examples. In addition, this concept of Addability and Subtractability contains the classical Insertability and Erasability of key value pairs as a special case so it provides a broader new semantics without loss of the *classical* one.

Aggregation is implemented for functions add and subtract by propagating a Combiner functor to combine associated values of type CodomainT. The standard Combiner is set as default template parameter template<class>class Combine = inplace_plus, which is again generically implemented by operator += for all Addable types.

For `Combine` functors, the Icl provides an `inverse` functor.

`Combine<T>`	`inverse<Combine<T> >::type`
`inplace_plus<T>`	`inplace_minus<T>`
`inplace_et<T>`	`inplace_caret<T>`
`inplace_star<T>`	`inplace_slash<T>`
`inplace_max<T>`	`inplace_min<T>`
`inplace_identity<T>`	`inplace_erasure<T>`
`Functor`	`inplace_erasure<T>`

The meta function `inverse` is mutually implemented for all but the default functor `Functor` such that e.g. `inverse<inplace_minus<T> >::type` yields `inplace_plus<T>`. Not in every case, e.g. `max/min`, does the `inverse` functor invert the effect of it's antetype. But for the default it does:

	`_add<Combine<CodomainT> >((k,x))`	`_subtract<inverse<Combine<CodomainT> >::type>((k,x))`
Instance	`_add<inplace_plus<int> >((k,x))`	`_subtract<inplace_minus<int> >((k,x))`
Inversion	adds `x` on overlap. This inverts a preceding `subtract` of `x` on `k`	subtracts `x` on overlap. This inverts a preceding `add` of `x` on `k`

As already mentioned aggregating `Addability` and `Subtractability` on `Maps` contains the *classical* `Insertability` and `Erasability` of key value pairs as a special case:

aggregating function	**equivalent *classical* function**
`_add<inplace_identity<CodomainT> >(const value_type&)`	`insert(const value_type&)`
`_subtract<inplace_erasure<CodomainT> >(const value_type&)`	`erase(const value_type&)`

The aggregating member function templates `_add` and `_subtract` are not in the public interface of `interval_maps`, because the `Combine` functor is intended to be an invariant of `interval_map's` template instance to avoid, that clients spoil the aggregation by accidentally calling varying aggregation functors. But you could instantiate an `interval_map` to have `insert/erase` semantics this way:

```
interval_map<int,int,partial_absorber,
             std::less,
             inplace_identity //Combine parameter specified
            > m;
interval<int>::type itv = interval<int>::rightopen(2,7);
m.add(make_pair(itv,42));      //same as insert
m.subtract(make_pair(itv,42)); //same as erase
```

This is, of course, only a clarifying example. Member functions `insert` and `erase` are available in `interval_map's` interface so they can be called directly.

Map Traits

Icl maps differ in their behavior dependent on how they handle ***identity elements*** of the associated type `CodomainT`.

Remarks on Identity Elements

In the pseudo code snippets below `0` will be used to denote `identity elements`, which can be different objects like `const double 0.0`, empty sets, empty strings, null-vectors etc. dependent of the instance type for parameter `CodomainT`. The existence of an ***identity element*** wrt. an `operator+=` is a requirement for template type `CodomainT`.

type	operation	identity element
`int`	addition	`0`
`string`	concatenation	`""`
`set<T>`	union	`{}`

In these cases the `identity element` value is delivered by the default constructor of the maps `CodomainT` type. But there are well known exceptions like e.g. numeric multiplication:

type	operation	identity element
`int`	multiplication	`1`

Therefore icl functors, that serve as `Combiner` parameters of icl Maps implement a static function `identity_element()` to make sure that the correct `identity_element()` is used in the implementation of *aggregate on overlap*.

```
inplace_times<int>::identity_element() == 1
// or more general
inplace_times<T>::identity_element() == unit_element<T>::value()
```

Definedness and Storage of Identity Elements

There are two *properties* or *traits* of icl maps that can be chosen by a template parameter `Traits`. The ***first trait*** relates to the ***definedness*** of the map. Icl maps can be **partial** or **total** on the set of values given by domain type `DomainT`.

- A ***partial*** map is only defined on those key elements that have been inserted into the Map. This is usually expected and so ***partial definedness*** is the default.
- Alternatively an icl Map can be ***total***. It is then considered to contain a ***neutral value*** for all key values that are not stored in the map.

The ***second trait*** is related to the representation of `identity elements` in the map. An icl map can be a ***identity absorber*** or a ***identity enricher***.

- A ***identity absorber*** never stores value pairs `(k,0)` that carry identity elements.
- A ***identity enricher*** stores value pairs `(k,0)`.

For the template parameter `Traits` of icl Maps we have the following four values.

	identity absorber	**identity enricher**
partial	partial_absorber *(default)*	partial_enricher
total	total_absorber	total_enricher

Map Traits motivated

Map traits are a late extension to the **icl**. Interval maps have been used for a couple of years in a variety of applications at Cortex Software GmbH with an implementation that resembled the default trait. Only the deeper analysis of the icl's ***aggregating Map's concept*** in the course of preparation of the library for boost led to the introduction of map Traits.

Add-Subtract Antinomy in Aggregating Maps

Constitutional for the absorber/enricher propery is a little antinomy.

We can insert value pairs to the map by ***adding*** them to the map via operations `add, +=` or `+`:

```
{} + {(k,1)} == {(k,1)} // addition
```

Further addition on common keys triggers aggregation:

```
{(k,1)} + {(k,1)} == {(k,2)} // aggregation for common key k
```

A subtraction of existing pairs

```
{(k,2)} - {(k,2)} == {(k,0)} // aggregation for common key k
```

yields value pairs that are associated with 0-values or `identity elements`.

So once a value pair is created for a key `k` it can not be removed from the map via subtraction (`subtract, -=` or `-`).

The very basic fact on sets, that we can remove what we have previously added

```
x - x = {}
```

does not apply.

This is the motivation for the ***identity absorber*** Trait. A identity absorber map handles value pairs that carry identity elements as ***non-existent***, which saves the law:

```
x - x = {}
```

Yet this introduces a new problem: With such a *identity absorber* we are *by definition* unable to store a value `(k,0)` in the map. This may be unfavorable because it is not inline with the behavior of stl::maps and this is not necessarily expected by clients of the library.

The solution to the problem is the introduction of the identity enricher Trait, so the user can choose a map variant according to her needs.

Partial and Total Maps

The idea of a identity absorbing map is, that an ***associated identity element*** value of a pair `(k,0)` ***codes non-existence*** for it's key `k`. So the pair `(k,0)` immediately tunnels from a map where it may emerge into the realm of non existence.

```
{(k,0)} == {}
```

If identity elements do not code ***non-existence*** but ***existence with null quantification***, we can also think of a map that has an associated identity element ***for every*** key `k` that has no associated value different from 0. So in contrast to modelling **all** neutral value pairs `(k,0)` as being ***non-existent*** we can model **all** neutral value pairs `(k,0)` as being ***implicitly existent***.

A map that is modelled in this way, is one large vector with a value `v` for every key `k` of it's domain type `DomainT`. But only non-identity values are actually stored. This is the motivation for the definedness-Trait on `icl Maps`.

A ***partial*** map models the intuitive view that only value pairs are existent, that are stored in the map. A ***total*** map exploits the possibility that all value pairs that are not stored can be considered as being existent and ***quantified*** with the identity element.

Pragmatical Aspects of Map Traits

From a pragmatic perspective value pairs that carry `identity elements` as mapped values can often be ignored. If we count, for instance, the number of overlaps of inserted intervals in an `interval_map` (see example overlap counter), most of the time, we are not interested in whether an overlap has been counted `0` times or has not been counted at all. A identity enricher map is only needed, if we want to distinct between non-existence and 0-quantification.

The following distinction can **not** be made for a `partial_absorber` map but it can be made for an `partial_enricher` map:

```
(k,v) does not exist in the map: Pair (k,v) has NOT been dealt with
(k,0) key k carries 0          : Pair (k,v) has     been dealt with resulting in v=0
```

Sometimes this subtle distinction is needed. Then a `partial_enricher` is the right choice. Also, If we want to give two `icl::Maps` a common set of keys in order to, say, iterate synchronously over both maps, we need *enrichers*.

Semantics

“Beauty is the ultimate defense against complexity” -- David Gelernter

In the **icl** we follow the notion, that the semantics of a ***concept*** or ***abstract data type*** can be expressed by ***laws***. We formulate laws over interval containers that can be evaluated for a given instantiation of the variables contained in the law. The following pseudocode gives a shorthand notation of such a law.

```
Commutativity<T,+>:
T a, b; a + b == b + a;
```

This can of course be coded as a proper c++ class template which has been done for the validation of the **icl**. For sake of simplicity we will use pseudocode here.

The laws that describe the semantics of the **icl's** class templates were validated using the Law based Test Automaton ***LaBatea***, a tool that generates instances for the law's variables and then tests it's validity. Since the **icl** deals with sets, maps and relations, that are well known objects from mathematics, the laws that we are using are mostly *recycled* ones. Also some of those laws are grouped in notions like e.g. *orderings* or *algebras*.

Orderings and Equivalences

Lexicographical Ordering and Equality

On all set and map containers of the icl, there is an `operator <` that implements a strict weak ordering. The semantics of `operator <` is the same as for an stl's SortedAssociativeContainer, specifically stl::set and stl::map:

```
Irreflexivity<T,< > : T a;     !(a<a)
Asymmetry<T,< >     : T a,b;   a<b implies !(b<a)
Transitivity<T,< >  : T a,b,c; a<b && b<c implies a<c
```

`Operator <` depends on the icl::container's template parameter `Compare` that implements a *strict weak ordering* for the container's `domain_type`. For a given `Compare` ordering, `operator <` implements a lexicographical comparison on icl::containers, that uses the `Compare` order to establish a unique sequence of values in the container.

The induced equivalence of `operator <` is lexicographical equality which is implemented as `operator ==`.

```
//equivalence induced by strict weak ordering <
!(a<b) && !(b<a) implies a == b;
```

Again this follows the semantics of the **stl**. Lexicographical equality is stronger than the equality of elements. Two containers that contain the same elements can be lexicographically unequal, if their elements are differently sorted. Lexicographical comparison belongs to the ***segmental*** aspect. Of all the different sequences that are valid for unordered sets and maps, one such sequence is selected by the `Compare` order of elements. Based on this selection a unique iteration is possible.

Subset Ordering and Element Equality

On the fundamental aspect only membership of elements matters, not their sequence. So there are functions `contained_in` and `element_equal` that implement the subset relation and the equality on elements. Yet, `contained_in` and `is_element_equal` functions are not really working on the level of elements. They also work on the basis of the containers templates `Compare` parameter. In practical terms we need to distinguish between lexicographical equality `operator ==` and equality of elements `is_element_equal`, if we work with interval splitting interval containers:

```
split_interval_set<time> w1, w2; //Pseudocode
w1 = {[Mon       ..       Sun)}; //split_interval_set containing a week
w2 = {[Mon .. Fri)[Sat .. Sun)}; //Same week split in work and week end parts.
w1 == w2;                        //false: Different segmentation
is_element_equal(w1,w2);         //true:  Same elements contained
```

For a constant `Compare` order on key elements, member function `contained_in` that is defined for all icl::containers implements a partial order on icl::containers.

```
with <= for contained_in,
    =e= for is_element_equal:
Reflexivity<T,<= >     : T a;     a <= a
Antisymmetry<T,<=,=e=> : T a,b;   a <= b && b <= a implies a =e= b
Transitivity<T,<= >    : T a,b,c; a <= b && b <= c implies a <= c
```

The induced equivalence is the equality of elements that is implemented via function `is_element_equal`.

```
//equivalence induced by the partial ordering contained_in on icl::container a,b
a.contained_in(b) && b.contained_in(a) implies is_element_equal(a, b);
```

Sets

For all set types `S` that are models concept `Set` (`std::set` , `interval_set`, `separate_interval_set` and `split_interval_set`) most of the well known mathematical laws on sets were successfully checked via LaBatea. The next tables are giving an overview over the checked laws ordered by operations. If possible, the laws are formulated with the stronger lexicographical equality (`operator ==`) which implies the law's validity for the weaker element equality `is_element_equal`. Throughout this chapter we will denote element equality as `=e=` instead of `is_element_equal` where a short notation is advantageous.

Laws on set union

For the operation ***set union*** available as `operator +, +=, |, |=` and the neutral element `identity_element<S>::value()` which is the empty set `S()` these laws hold:

```
Associativity<S,+,== >: S a,b,c; a+(b+c) == (a+b)+c
Neutrality<S,+,== >   : S a;       a+S() == a
Commutativity<S,+,== >: S a,b;       a+b == b+a
```

Laws on set intersection

For the operation ***set intersection*** available as `operator &, &=` these laws were validated:

```
Associativity<S,&,== >: S a,b,c; a&(b&c) == (a&b)&c
Commutativity<S,&,== >: S a,b;       a&b == b&a
```

Laws on set difference

For set difference there are only these laws. It is not associative and not commutative. It's neutrality is non symmetrical.

```
RightNeutrality<S,-,== > : S a;   a-S() == a
Inversion<S,-,== >:        S a;   a - a == S()
```

Summarized in the next table are laws that use `+`, `&` and `-` as a single operation. For all validated laws, the left and right hand sides of the equations are lexicographically equal, as denoted by `==` in the cells of the table.

```
                 +     &    -
Associativity    ==    ==
Neutrality       ==         ==
Commutativity    ==    ==
Inversion                   ==
```

Distributivity Laws

Laws, like distributivity, that use more than one operation can sometimes be instantiated for different sequences of operators as can be seen below. In the two instantiations of the distributivity laws operators `+` and `&` are swapped. So we can have small operator signatures like `+,&` and `&,+` to describe such instantiations, which will be used below. Not all instances of distributivity laws hold for lexicographical equality. Therefore they are denoted using a *variable* equality `=v=` below.

```
     Distributivity<S,+,&,=v= > : S a,b,c; a + (b & c) =v= (a + b) & (a + c)
     Distributivity<S,&,+,=v= > : S a,b,c; a & (b + c) =v= (a & b) + (a & c)
RightDistributivity<S,+,-,=v= > : S a,b,c; (a + b) - c =v= (a - c) + (b - c)
RightDistributivity<S,&,-,=v= > : S a,b,c; (a & b) - c =v= (a - c) & (b - c)
```

The next table shows the relationship between law instances, interval combining style and the used equality relation.

```
                                 +,&      &,+
     Distributivity  joining     ==       ==
                     separating  ==       ==
                     splitting   =e=      =e=

                                 +,-      &,-
RightDistributivity  joining     ==       ==
                     separating  ==       ==
                     splitting   =e=      ==
```

The table gives an overview over 12 instantiations of the four distributivity laws and shows the equalities which the instantiations holds for. For instance `RightDistributivity` with operator signature `+,-` instantiated for `split_interval_sets` holds only for element equality (denoted as `=e=`):

```
RightDistributivity<S,+,-,=e= > : S a,b,c; (a + b) - c =e= (a - c) + (b - c)
```

The remaining five instantiations of `RightDistributivity` are valid for lexicographical equality (demoted as `==`) as well.

Interval combining styles correspond to containers according to

```
style       set
joining     interval_set
separating  separate_interval_set
splitting   split_interval_set
```

Finally there are two laws that combine all three major set operations: De Mogans Law and Symmetric Difference.

DeMorgan's Law

De Morgans Law is better known in an incarnation where the unary complement operation `~` is used. `~(a+b) == ~a * ~b`. The version below is an adaption for the binary set difference `-`, which is also called ***relative complement***.

```
DeMorgan<S,+,&,=v= > : S a,b,c; a - (b + c) =v= (a - b) & (a - c)
DeMorgan<S,&,+,=v= > : S a,b,c; a - (b & c) =v= (a - b) + (a - c)
```

```
                              +,&      &,+
DeMorgan  joining             ==       ==
          separating          ==       =e=
          splitting           ==       =e=
```

Again not all law instances are valid for lexicographical equality. The second instantiations only holds for element equality, if the interval sets are non joining.

Symmetric Difference

```
SymmetricDifference<S,== > : S a,b,c; (a + b) - (a & b) == (a - b) + (b - a)
```

Finally Symmetric Difference holds for all of icl set types and lexicographical equality.

Maps

By definition a map is set of pairs. So we would expect maps to obey the same laws that are valid for sets. Yet the semantics of the **icl's** maps may be a different one, because of it's aggregating facilities, where the aggregating combiner operations are passed to combine the map's associated values. It turns out, that the aggregation on overlap principle induces semantic properties to icl maps in such a way, that the set of equations that are valid will depend on the semantics of the type `CodomainT` of the map's associated values.

This is less magical as it might seem at first glance. If, for instance, we instantiate an `interval_map` to collect and concatenate `std::strings` associated to intervals,

```
interval_map<int,std::string> cat_map;
cat_map += make_pair(interval<int>::rightopen(1,5),std::string("Hello"));
cat_map += make_pair(interval<int>::rightopen(3,7),std::string(" World"));
cout << "cat_map: " << cat_map << endl;
```

we won't be able to apply `operator -=`

```
// This will not compile because string::operator -= is missing.
cat_map -= make_pair(interval<int>::rightopen(3,7),std::string(" World"));
```

because, as std::sting does not implement -= itself, this won't compile. So all **laws**, that rely on `operator -=` or - not only will not be valid they can not even be stated. This reduces the set of laws that can be valid for a richer `CodomainT` type to a smaller set of laws and thus to a less restricted semantics.

Currently we have investigated and validated two major instantiations of icl::Maps,

- ***Maps of Sets*** that will be called ***Collectors*** and
- ***Maps of Numbers*** which will be called ***Quantifiers***

both of which seem to have many interesting use cases for practical applications. The semantics associated with the term *Numbers* is a commutative monoid for unsigned numbers and a commutative or abelian group for signed numbers. From a practical point of view we can think of numbers as counting or quantifying the key values of the map.

Icl ***Maps of Sets*** or ***Collectors*** are models of concept `Set`. This implies that all laws that have been stated as a semantics for `icl::Sets` in the previous chapter also hold for `Maps of Sets`. Icl ***Maps of Numbers*** or ***Quantifiers*** on the contrary are not models of concept `Set`. But there is a substantial intersection of laws that apply both for `Collectors` and `Quantifiers`.

Kind of Map	Alias	Behavior
Maps of Sets	Collector	Collects items **for** key values
Maps of Numbers	Quantifier	Counts or quantifies **the** key values

In the next two sections the law based semantics of ***Collectors*** and ***Quantifiers*** will be described in more detail.

Collectors: Maps of Sets

Icl `Collectors`, behave like `Sets`. This can be understood easily, if we consider, that every map of sets can be transformed to an equivalent set of pairs. For instance in the pseudocode below map `m`

```
icl::map<int,set<int> >  m = {(1->{1,2}), (2->{1})};
```

is equivalent to set `s`

```
icl::set<pair<int,int> > s = {(1,1),(1,2),   //representing 1->{1,2}
                              (2,1)       }; //representing 2->{1}
```

Also the results of add, subtract and other operations on `map m` and `set s` preserves the equivalence of the containers ***almost*** perfectly:

```
m += (1,3);
m == {(1->{1,2,3}), (2->{1})}; //aggregated on collision of key value 1
s += (1,3);
s == {(1,1),(1,2),(1,3),   //representing 1->{1,2,3}
      (2,1)             }; //representing 2->{1}
```

The equivalence of `m` and `s` is only violated if an empty set occurres in `m` by subtraction of a value pair:

```
m -= (2,1);
m == {(1->{1,2,3}), (2->{})}; //aggregated on collision of key value 2
s -= (2,1);
s == {(1,1),(1,2),(1,3)   //representing 1->{1,2,3}
                        }; //2->{} is not represented in s
```

This problem can be dealt with in two ways.

1. Deleting value pairs form the Collector, if it's associated value becomes a neutral value or `identity_element`.
2. Using a different equality, called distinct equality in the laws to validate. Distinct equality only accounts for value pairs that that carry values unequal to the `identity_element`.

Solution (1) led to the introduction of map traits, particularly trait ***partial_absorber***, which is the default setting in all icl's map templates.

Solution (2), is applied to check the semantics of icl::Maps for the partial_enricher trait that does not delete value pairs that carry identity elements. Distinct equality is implemented by a non member function called `is_distinct_equal`. Throughout this chapter distinct equality in pseudocode and law denotations is denoted as `=d=` operator.

The validity of the sets of laws that make up `Set` semantics should now be quite evident. So the following text shows the laws that are validated for all `Collector` types `C`. Which are `icl::map`<D,S,T>, `interval_map`<D,S,T> and `split_interval_map`<D,S,T> where `CodomainT` type `S` is a model of `Set` and `Trait` type `T` is either `partial_absorber` or `partial_enricher`.

Laws on set union, set intersection and set difference

```
Associativity<C,+,== >: C a,b,c; a+(b+c) == (a+b)+c
Neutrality<C,+,== >   : C a;       a+C() == a
Commutativity<C,+,== >: C a,b;       a+b == b+a

Associativity<C,&,== >: C a,b,c; a&(b&c) ==(a&b)&c
Commutativity<C,&,== >: C a,b;       a&b == b&a

RightNeutrality<C,-,== >: C a;   a-C() ==  a
Inversion<C,-,=v= >     : C a;   a - a =v= C()
```

All the fundamental laws could be validated for all icl Maps in their instantiation as Maps of Sets or Collectors. As expected, Inversion only holds for distinct equality, if the map is not a `partial_absorber`.

```
                             +    &    -
Associativity                ==   ==
Neutrality                   ==        ==
Commutativity                ==   ==
Inversion partial_absorber             ==
          partial_enricher             =d=
```

Distributivity Laws

```
     Distributivity<C,+,&,=v= > : C a,b,c; a + (b & c) =v= (a + b) & (a + c)
     Distributivity<C,&,+,=v= > : C a,b,c; a & (b + c) =v= (a & b) + (a & c)
RightDistributivity<C,+,-,=v= > : C a,b,c; (a + b) - c =v= (a - c) + (b - c)
RightDistributivity<C,&,-,=v= > : C a,b,c; (a & b) - c =v= (a - c) & (b - c)
```

Results for the distributivity laws are almost identical to the validation of sets except that for a `partial_enricher map` the law `(a & b) - c == (a - c) & (b - c)` holds for lexicographical equality.

```
                                                 +,&    &,+
     Distributivity  joining                     ==     ==
                     splitting  partial_absorber =e=    =e=
                                partial_enricher =e=    ==

                                                 +,-    &,-
RightDistributivity  joining                     ==     ==
                     splitting                   =e=    ==
```

DeMorgan's Law and Symmetric Difference

```
DeMorgan<C,+,&,=v= > : C a,b,c; a - (b + c) =v= (a - b) & (a - c)
DeMorgan<C,&,+,=v= > : C a,b,c; a - (b & c) =v= (a - b) + (a - c)
```

```
                         +,&      &,+
DeMorgan  joining        ==       ==
          splitting      ==       =e=
```

```
SymmetricDifference<C,== > : C a,b,c; (a + b) - (a * b) == (a - b) + (b - a)
```

Reviewing the validity tables above shows, that the sets of valid laws for `icl Sets` and `icl Maps of Sets` that are *identity absorbing* are exactly the same. As expected, only for Maps of Sets that represent empty sets as associated values, called *identity enrichers*, there are marginal semantic differences.

Quantifiers: Maps of Numbers

Subtraction on Quantifiers

With `Sets` and `Collectors` the semantics of `operator -` is that of *set difference* which means, that you can only subtract what has been put into the container before. With `Quantifiers` that ***count*** or ***quantify*** their key values in some way, the semantics of `operator -` may be different.

The question is how subtraction should be defined here?

```
//Pseudocode:
icl::map<int,some_number> q = {(1->1)};
q -= (2->1);
```

If type `some_number` is `unsigned` a *set difference* kind of subtraction make sense

```
icl::map<int,some_number> q = {(1->1)};
q -= (2->1);   // key 2 is not in the map so
q == {(1->1)}; // q is unchanged by 'aggregate on collision'
```

If `some_number` is a `signed` numerical type the result can also be this

```
icl::map<int,some_number> q = {(1->1)};
q -= (2->1);                // subtracting works like
q == {(1->1), (2-> -1)}; // adding the inverse element
```

As commented in the example, subtraction of a key value pair `(k,v)` can obviously be defined as adding the ***inverse element*** for that key `(k,-v)`, if the key is not yet stored in the map.

Partial and Total Quantifiers and Infinite Vectors

Another concept, that we can think of, is that in a `Quantifier` every `key_value` is initially quantified `0`-times, where `0` stands for the neutral element of the numeric `CodomainT` type. Such a `Quantifier` would be totally defined on all values of it's `DomainT` type and can be conceived as an `InfiniteVector`.

To create an infinite vector that is totally defined on it's domain we can set the map's `Trait` parameter to the value `total_absorber`. The `total_absorber` trait fits specifically well with a `Quantifier` if it's `CodomainT` has an inverse element, like all signed numerical type have. As we can see later in this section this kind of a total `Quantifier` has the basic properties that elements of a vector space do provide.

Intersection on Quantifiers

Another difference between `Collectors` and `Quantifiers` is the semantics of `operator &`, that has the meaning of set intersection for `Collectors`.

For the *aggregate on overlap principle* the operation `&` has to be passed to combine associated values on overlap of intervals or collision of keys. This can not be done for `Quantifiers`, since numeric types do not implement intersection.

For `CodomainT` types that are not models of `Sets` `operator &` is defined as *aggregation on the intersection of the domains*. Instead of the `codomain_intersect` functor `codomain_combine` is used as aggregation operation:

```
//Pseudocode example for partial Quantifiers p, q:
interval_map<int,int> p, q;
p     = {[1     3)->1   };
q     = {   ([2    4)->1};
p & q =={    [2 3)->2   };
```

So an addition or aggregation of associated values is done like for `operator +` but value pairs that have no common keys are not added to the result.

For a `Quantifier` that is a model of an `InfiniteVector` and which is therefore defined for every key value of the `DomainT` type, this definition of `operator &` degenerates to the same sematics that `operaotor +` implements:

```
//Pseudocode example for total Quantifiers p, q:
interval_map<int,int> p, q;
p   = {[min   1)[1      3)[3         max]};
          ->0      ->1         ->0
q   = {[min        2)[2      4)[4    max]};
            ->0        ->1       ->0
p&q =={[min   1)[1  2)[2 3)[3 4)[4   max]};
          ->0    ->1    ->2  ->1    ->0
```

Laws for Quantifiers of unsigned Numbers

The semantics of icl Maps of Numbers is different for unsigned or signed numbers. So the sets of laws that are valid for Quantifiers will be different depending on the instantiation of an unsigned or a signed number type as `CodomainT` parameter.

Again, we are presenting the investigated sets of laws, this time for `Quantifier` types `Q` which are `icl::map<D,N,T>`, `interval_map<D,N,T>` and `split_interval_map<D,N,T>` where `CodomainT` type `N` is a `Number` and `Trait` type `T` is one of the icl's map traits.

```
Associativity<Q,+,== >: Q a,b,c; a+(b+c) == (a+b)+c
Neutrality<Q,+,== >    : Q a;       a+Q() == a
Commutativity<Q,+,== >: Q a,b;        a+b == b+a

Associativity<Q,&,== >: Q a,b,c; a&(b&c) ==(a&b)&c
Commutativity<Q,&,== >: Q a,b;        a&b == b&a

RightNeutrality<Q,-,== >: Q a;   a-Q() ==  a
Inversion<Q,-,=v= >     : Q a;   a - a =v= Q()
```

For an `unsigned Quantifier`, an icl Map of `unsigned numbers`, the same basic laws apply that are valid for `Collectors`:

```
                               +    &    -
Associativity                  ==   ==
Neutrality                     ==        ==
Commutativity                  ==   ==
Inversion absorbs_identities             ==
          enriches_identities            =d=
```

The subset of laws, that relates to `operator +` and the neutral element `Q()` is that of a commutative monoid. This is the same concept, that applies for the `CodomainT` type. This gives rise to the assumption that an icl `Map` over a `CommutativeModoid` is again a `CommutativeModoid`.

Other laws that were valid for `Collectors` are not valid for an `unsigned Quantifier`.

Laws for Quantifiers of signed Numbers

For `Quantifiers` of signed numbers, or `signed Quantifiers`, the pattern of valid laws is somewhat different:

```
                              +    &    -
Associativity                 =v=  =v=
Neutrality                    ==        ==
Commutativity                 ==   ==
Inversion absorbs_identities            ==
          enriches_identities           =d=
```

The differences are tagged as =v= indicating, that the associativity law is not uniquely valid for a single equality relation == as this was the case for `Collector` and `unsigned Quntifier` maps.

The differences are these:

```
                                +
Associativity        icl::map   ==
                 interval_map   ==
           split_interval_map   =e=
```

For `operator +` the associativity on `split_interval_maps` is only valid with element equality =e=, which is not a big constrained, because only element equality is required.

For `operator &` the associativity is broken for all maps that are partial absorbers. For total absorbers associativity is valid for element equality. All maps having the *identity enricher* Trait are associative wrt. lexicographical equality ==.

```
Associativity                    &
   absorbs_identities && !is_total   false
   absorbs_identities &&  is_total   =e=
   enriches_identities               ==
```

Note, that all laws that establish a commutative monoid for `operator +` and identity element `Q()` are valid for `signed Quantifiers`. In addition symmetric difference that does not hold for `unsigned Qunatifiers` is valid for `signed Qunatifiers`.

```
SymmetricDifference<Q,== > : Q a,b,c; (a + b) - (a & b) == (a - b) + (b - a)
```

For a `signed TotalQuantifier Qt` symmetrical difference degenerates to a trivial form since `operator &` and `operator +` become identical

```
SymmetricDifference<Qt,== > : Qt a,b,c; (a + b) - (a + b) == (a - b) + (b - a) == Qt()
```

Existence of an Inverse

By now `signed Quantifiers Q` are commutative monoids with respect to the `operator +` and the neutral element `Q()`. If the Quantifier's `CodomainT` type has an *inverse element* like e.g. `signed numbers` do, the `CodomainT` type is a ***commutative*** or ***abelian group***. In this case a `signed Quantifier` that is also ***total*** has an ***inverse*** and the following law holds:

```
InverseElement<Qt,== > : Qt a; (0 - a) + a == 0
```

Which means that each `TotalQuantifier` over an abelian group is an abelian group itself.

This also implies that a `Quantifier` of `Quantifiers` is again a `Quantifiers` and a `TotalQuantifier` of `TotalQuantifiers` is also a `TotalQuantifier`.

`TotalQuantifiers` resemble the notion of a vector space partially. The concept could be completed to a vector space, if a scalar multiplication was added.

Concept Induction

Obviously we can observe the induction of semantics from the `CodomainT` parameter into the instantiations of icl maps.

	is model of	**if**	**example**
`Map<D,Monoid>`	`Modoid`		`interval_map<int,string>`
`Map<D,Set,Trait>`	`Set`	`Trait::absorbs_identities`	`interval_map<int,std::set<int> >`
`Map<D,CommutativeMonoid>`	`CommutativeMonoid`		`interval_map<int,unsigned int>`
`Map<D,CommutativeGroup>`	`CommutativeGroup`	`Trait::is_total`	`interval_map<int,int,total_absorber>`

Interface

Section **Interface** outlines types and functions of the **Icl**. Synoptical tables allow to review the overall structure of the libraries design and to focus on structural equalities and differences with the corresponding containers of the standard template library.

Class templates

Intervals

In the **icl** we have two groups of interval types. There are ***statically bounded*** intervals, `right_open_interval`, `left_open_interval`, `closed_interval`, `open_interval`, that always have the same kind of interval borders and ***dynamically bounded*** intervals, `discrete_interval`, `continuous_interval` which can have one of the four possible bound types at runtime.

Table 6. Interval class templates

group	form	template	instance parameters
statically bounded	asymmetric	`right_open_interval`	`<class DomainT, template<class>class Compare>`
		`left_open_interval`	`<...same for all interval class templates...>`
	symmetric	`closed_interval`	
		`open_interval`	
dynamically bounded		`discrete_interval`	
		`continuous_interval`	

Not every class template works with all domain types. Use interval class templates according the next table.

Table 7. Usability of interval class templates for discrete or continuous domain types

group	form	template	discrete	continuous
statically bounded	asymmetric	`right_open_interval`	yes	yes
		`left_open_interval`	yes	yes
	symmetric	`closed_interval`	yes	
		`open_interval`	yes	
dynamically bounded		`discrete_interval`	yes	
		`continuous_interval`		yes

From a pragmatical point of view, the most important interval class template of the *statically bounded* group is `right_open_interval`. For discrete domain types also closed intervals might be convenient. Asymmetric intervals can be used with continuous domain types but `continuous_interval` is the only class template that allows to represent a singleton interval that contains only one element.

Use `continuous_interval`, if you work with interval containers of countinuous domain types and you want to be able to handle single values:

```
typedef interval_set<std::string, std::less, continuous_interval<std::string> > IdentifiersT;
IdentifiersT identifiers, excluded;
identifiers += continuous_interval<std::string>::right_open("a", "c");

// special identifiers shall be excluded
identifiers -= std::string("boost");
cout << "identifiers: " << identifiers << endl;

excluded = IdentifiersT(icl::hull(identifiers)) - identifiers;
cout << "excluded   : " << excluded << endl;

//------ Program output: --------
identifiers: {[a,boost)(boost,c)}
excluded   : {[boost,boost]}
```

Library defaults and class template `interval`

As shown in the example above, you can choose an interval type by instantiating the interval container template with the desired type.

```
typedef interval_set<std::string, std::less, continuous_interval<std::string> > IdentifiersT;
```

But you can work with the library default for interval template parameters as well, which is `interval<DomainT,Compare>::type`.

	interval bounds	domain_type	interval_default
# i f d e f BOOST_ICL_USE_STATIC_BOUNDED_INTERVALS	static		`right_open_interval`
`#else`	dynamic	discrete	`discrete_interval`
		continuous	`continuous_interval`

So, if you are always happy with the library default for the interval type, just use

```
icl::interval<MyDomainT>::type myInterval;
```

as you standard way of declaring intervals and default parameters for interval containers:

```
typedef interval_set<std::string> IdentifiersT;
IdentifiersT identifiers, excluded;
identifiers += interval<std::string>::right_open("a", "c");
. . .
```

So class template `interval` provides a standard way to work with the library default for intervals. Via `interval<D,C>::type` you can declare a default interval. In addition four static functions

```
T interval<D,C>::right_open(const D&, const D&);
T interval<D,C>::left_open(const D&, const D&);
T interval<D,C>::closed(const D&, const D&);
T interval<D,C>::open(const D&, const D&);
```

allow to construct intervals of the library default `T = interval<D,C>::type`.

If you

```
#define BOOST_ICL_USE_STATIC_BOUNDED_INTERVALS
```

the library uses only statically bounded `right_open_interval` as default interval type. In this case, the four static functions above are also available, but they only move interval borders consistently, if their domain type is discrete, and create an appropriate `right_open_interval` finally:

```
interval<D,C>::right_open(a,b) == [a, b)  ->  [a  , b  )
interval<D,C>:: left_open(a,b) == (a, b]  ->  [a++, b++)
interval<D,C>::    closed(a,b) == [a, b]  ->  [a  , b++)
interval<D,C>::      open(a,b) == (a, b)  ->  [a++, b  )
```

For continuous domain types only the first of the four functions is applicable that matches the library default for statically bounded intervals: `right_open_interval`. The other three functions can not perform an appropriate tranformation and will not compile.

Sets

The next two tables give an overview over ***set class templates*** of the icl.

Table 8. Set class templates

group	template	instance parameters
`interval_sets`	`interval_set`	`<DomainT,Compare,IntervalT,Alloc>`
	`separate_interval_set`	`<DomainT,Compare,IntervalT,Alloc>`
	`split_interval_set`	`<DomainT,Compare,IntervalT,Alloc>`

Templates and template parameters, given in the preceding table are described in detail below. `Interval_sets` represent three class templates `interval_set`, `separate_interval_set` and `split_interval_set` that all have equal template parameters.

Table 9. Parameters of set class templates

	type of elements	order of elements	type of intervals	memory allocation
template parameter	`class`	`template <class>class`	`class`	`template <class>class`
`interval`	`DomainT`	`Compare = std::less`		
`interval_sets`	`DomainT`	`Compare = std::less`	`IntervalT = interval<DomainT,Compare>::type`	`Alloc = std::alloc`

Maps

The next two tables give an overview over ***map class templates*** of the icl.

Table 10. map class templates

group	template	instance parameters
`interval_maps`	`interval_map`	`<DomainT,CodomainT,Traits,Compare,Combine,Section,IntervalT,Alloc>`
	`split_interval_map`	`<DomainT,CodomainT,Traits,Compare,Combine,Section,IntervalT,Alloc>`
`icl::map`	`icl::map`	`<DomainT,CodomainT,Traits,Compare,Combine,Section,Alloc>`

Templates and template parameters, given in the preceding table are described in detail below. `Interval_maps` represent two class templates `interval_map` and `split_interval_map` that all have equal template parameters.

Table 11. Parameters of map class templates

	elements	mapped values	traits	order of elements	aggregation propagation	intersection propagation	type of intervals	memory allocation
template parameter	class	class	class	template <class>class	template <class>class	template <class>class	class	template <class>class
interval_maps	DomainT	CodomainT	Traits = identity_absorber	Compare = std::less	Combine = inplace_plus	Section = icl::inplace_et	IntervalT = interval<DomainT,Compare>::type	Alloc = std::alloc
icl::map	DomainT	CodomainT	Traits = identity_absorber	Compare = std::less	Combine = inplace_plus	Section = icl::inplace_et	Alloc = std::alloc	

Using the following placeholders,

```
D  := class DomainT,
C  := class CodomainT,
T  := class Traits,
cp := template<class D>class Compare = std::less,
cb := template<class C>class Combine = icl::inplace_plus,
s  := template<class C>class Section = icl::inplace_et,
I  := class IntervalT = icl::interval<D,cp>::type
a  := template<class>class Alloc = std::allocator
```

we arrive at a final synoptical matrix of class templates and their parameters.

```
interval     <D,       cp,             >
interval_sets<D,       cp,          I, a >
interval_maps<D, C, T, cp, cb, s, I, a >
icl::map     <D, C, T, cp, cb, s,     a >
```

The choice of parameters and their positions follow the std::containers as close a possible, so that usage of interval sets and maps does only require minimal additional knowledge.

Additional knowledge is required when instantiating a comparison parameter `Compare` or an allocation parameter `Alloc`. In contrast to std::containers these have to be instantiated as templates, like e.g.

```
interval_set<string, german_compare>      sections; // 2nd parameter is a template
std::set<string, german_compare<string> > words;    // 2nd parameter is a type
```

Required Concepts

There are uniform requirements for the template parameters across **icl's** class templates. The template parameters can be grouped with respect to those requirements.

	used in	Kind	Parameter	Instance	Description
Domain order	`Intervals, Sets, Maps`	`typename`	`DomainT`		For the type `DomainT` of key elements `...`
		`template`	`Compare`	`Compare<DomainT>`	`...` there is an order `Compare`
Interval type	`interval_sets/maps`	`typename`	`IntervalT`		`...` the `IntervalT` parameter has to use the same element type and order.
Codomain aggregation	`Maps`	`typename`	`CodomainT`		For the type `CodomainT` of associated values `...`
		`template`	`Combine`	`Combine<CodomainT>`	`...` there is a binary functor `Combine<CodomainT>()` to combine them
				`Inverse<Combine<CodomainT>>`	`...` and implicitly an `Inverse` functor to inversely combine them.
		`template`	`Section`	`Section<CodomainT>`	Intersection is propagated to CodomainT values via functor `Section<CodomainT>()`
Memory allocation	`Sets, Maps`	`template`	`Alloc`	`Alloc<`*various*`>`	An allocator can be chosen for memory allocation.

Requirements on DomainT

The next table gives an overview over the requirements for template parameter `DomainT`. Some requirements are dependent on *conditions*. Column *operators* shows the operators and functions that are expected for `DomainT`, if the default order `Compare = std::less` is used.

Parameter	Condition	Operators	Requirement
DomainT		DomainT(), <	Regular<DomainT> && StrictWeakOrdering<DomainT,Compare>
		++, unit_element<CodomainT>::value()	&& (IsIncrementable<DomainT>||HasUnitElement<DomainT>)
	IsIntegral<DomainT>	++, --	IsIncrementable<DomainT> && IsDecrementable<DomainT>

A domain type DomainT for intervals and interval containers has to satisfy the requirements of concept Regular which implies among other properties the existence of a copy and a default constructor. In addition IsIncrementable **or** HasUnitElement is required for DomainT. In the **icl** we represent an empty closed interval as interval [b,a] where a < b (here < represents Compare<DomainT>()). To construct one of these empty intervals as default constructor for any type DomainT we choose [1,0], where 0 is a null-value or identity_element and 1 is a one-value or unit_element:

```
interval() := [unit_element<DomainT>::value(), identity_element<DomainT>::value()] //pseudocode
```

Identity_elements are implemented via call of the default constructor of DomainT. A unit_element<T>::value() is implemented by default as a identity_element, that is incremented once.

```
template <class Type>
inline Type unit_element<Type>::value(){ return succ(identity_element<Type>::value()); };
```

So a type DomainT that is incrementable will also have an unit_element. If it does not, a unit_element can be provided. A unit_element can be any value, that is greater as the identity_element in the Compare order given. An example of a type, that has an identity_element but no increment operation is string. So for std::string a unit_element is implemented like this:

```
// Smallest 'visible' string that is greater than the empty string.
template <>
inline std::string unit_element<std::string>::value(){ return std::string(" "); };
```

Just as for the key type of std::sets and maps template parameter Compare is required to be a strict weak ordering on DomainT.

Finally, if DomainT is an integral type, DomainT needs to be incrementable and decrementable. This *'bicrementability'* needs to be implemented on the smallest possible unit of the integral type. This seems like being trivial but there are types like e.g. boost::date_time::ptime, that are integral in nature but do not provide the required in- and decrementation on the least incrementable unit. For icl::intervals incementation and decementation is used for computations between open to closed interval borders like e.g. [2,43) == [2,42]. Such computations always need only one in- or decrementation, if DomainT is an integral type.

Requirements on IntervalT

Requirements on the IntervalT parameter are closely related to the DomainT parameter. IntervalT has two associated types itself for an element type and a compare order that have to be consistent with the element and order parameters of their interval containers. IntervalT then has to implement an order called exclusive_less. Two intervals x, y are exclusive_less

```
icl::exclusive_less(x, y)
```

if all DomainT elements of x are less than elements of y in the Compare order.

Parameter	Operators	Requirement
IntervalT	exclusive_less	IsExclusiveLessComparable<Interval<DomainT,Compare> >

Requirements on CodomainT

Summarized in the next table are requirements for template parameter CodomainT of associated values for Maps. Again there are *conditions* for some of the requirements. Column *operators* contains the operators and functions required for CodomainT, if we are using the default combiner Combine = icl::inplace_plus.

Parameter	Condition	Operators	Requirement
CodomainT	add, subtract, intersect unused	CodomainT(), ==	Regular<CodomainT> which implies
			DefaultConstructible<CodomainT> && EqualityComparable<CodomainT>
	only add used	+=	&& Combinable<CodomainT,Combine>
	... and also subtract used	-=	&& Combinable<CodomainT,Inverse<Combine> >
	Section used and CodomainT is a set	&=	&& Intersectable<CodomainT,Section>

The requirements on the type CodomainT of associated values for a icl::map or interval_map depend on the usage of their aggregation functionality. If aggregation on overlap is never used, that is to say that none of the addition, subtraction and intersection operations (+, +=, add, -, -=, subtract, &, &=, add_intersection) are used on the interval_map, then CodomainT only needs to be Regular.

Regular object semantics implies DefaultConstructible and EqualityComparable which means it has a default ctor CodomainT() and an operator ==.

Use interval_maps ***without aggregation***, if the associated values are not addable but still are attached to intervals so you want to use interval_maps to handle them. As long as those values are added with insert and deleted with erase interval_maps will work fine with such values.

If ***only addition*** is used via interval_map's +, += or add but no subtraction, then CodomainT need to be Combinable for functor template Combine. That means in most cases when the default implementation inplace_plus for Combine is used, that CodomainT has to implement operator +=.

For associated value types, that are addable but not subtractable like e.g. std::string it usually makes sense to use addition to combine values but the inverse combination is not desired.

```
interval_map<int,std::string> cat_map;
cat_map += make_pair(interval<int>::rightopen(1,5),std::string("Hello"));
cat_map += make_pair(interval<int>::rightopen(3,7),std::string(" world"));
cout << "cat_map: " << cat_map << endl;
//cat_map: {([1,3)->Hello)([3,5)->Hello world)([5,7)-> world)}
```

For *complete aggregation functionality* an inverse aggregation functor on a `Map`'s `CodomainT` is needed. The icl provides a metafunction `inverse` for that purpose. Using the default `Combine = inplace_plus` that relies on the existence of `operator +=` on type `CodomainT` metafunction `inverse` will infer `inplace_minus` as inverse functor, that requires `operator -=` on type `CodomainT`.

In the icl's design we make the assumption, in particular for the default setting of parameters `Combine = inplace_plus`, that type `CodomainT` has a neutral element or `identity_element` with respect to the `Combine` functor.

Associated Types

In order to give an overview over ***associated types*** the **icl** works with, we will apply abbreviations again that were introduced in the presentaiton of icl class templates,

```
interval     <D,       cp,              >
interval_sets<D,       cp,          I, a >
interval_maps<D, C, T, cp, cb, s, I, a >
icl::map     <D, C, T, cp, cb, s,    a >
```

where these placeholders were used:

```
D  := class DomainT,
C  := class CodomainT,
T  := class Traits,
cp := template<class D>class Compare = std::less,
cb := template<class C>class Combine = icl::inplace_plus,
s  := template<class C>class Section = icl::inplace_et,
I  := class Interval = icl::interval<D,cp>::type
a  := template<class>class Alloc = std::allocator
```

With some additions,

```
sz := template<class D>class size
df := template<class D>class difference
Xl := class ExclusiveLess = exclusive_less<Interval<DomainT,Compare> >
inv:= template<class Combiner>class inverse
(T,U) := std::pair<T,U> for typnames T,U
```

we can summarize the associated types as follows. Again two additional columns for easy comparison with stl sets and maps are provided.

Table 12. Icl Associated types

Purpose	**Aspect**	**Type**	**intervals**	**interval sets**	**interval maps**	**element sets**	**element maps**
Data	fundamental	domain_type	D	D	D	D	D
		codomain_type	D	D	C	D	C
		element_type	D	D	(D,C)	D	(D,C)
		segment_type	i<D,cp>	i<D,cp>	(i<D,cp>,C)		
	size	size_type	sz<D>	sz<D>	sz<D>	sz<D>	sz<D>
		difference_type	df<D>	df<D>	df<D>	sz<D>	sz<D>
			intervals	interval sets	interval maps	element sets	element maps
Data	segmental	key_type	D	i<D,cp>	i<D,cp>	D	D
		data_type	D	i<D,cp>	C	D	C
		value_type	D	i<D,cp>	(i<D,cp>,C)	D	(D,C)
		interval_type	i<D,cp>	i<D,cp>	i<D,cp>		
	allocation	allocator_type		a<i<D,cp>>	a<(i<D,cp>, C)>	a<D>	a<(D,C)>
			intervals	interval sets	interval maps	element sets	element maps
Ordering	fundamental	domain_compare	cp<D>	cp<D>	cp<D>	cp<D>	cp<D>
	segmental	key_compare	cp<D>	Xl	Xl	cp<D>	cp<D>
		interval_compare		Xl	Xl		
Aggregation	fundamental	codomain_combine			cb<C>		cb<C>

Purpose	Aspect	Type	intervals	interval sets	interval maps	element sets	element maps
		`inverse_codomain_combine`			`inv<cb<C>>`		`inv<cb<C>>`
		`codomain_intersect`			`s<C>`		`s<C>`
		`inverse_codomain_intersect`			`inv<s<C>>`		`inv<s<C>>`

Function Synopsis

In this section a single ***matrix*** is given, that shows all ***functions*** with shared names and identical or analogous semantics and their polymorphical overloads across the class templates of the **icl**. In order to achieve a concise representation, a series of ***placeholders*** are used throughout the function matrix.

The ***placeholder's*** purpose is to express the polymorphic usage of the functions. The ***first column*** of the function matrix contains the signatures of the functions. Within these signatures `T` denotes a container type and `J` and `P` polymorphic argument and result types.

Within the body of the matrix, sets of **boldface** placeholders denote the sets of possible instantiations for a polymorphic placeholder `P`. For instance **e i S** denotes that for the argument type `P`, an element **e**, an interval **i** or an interval_set **S** can be instantiated.

If the polymorphism can not be described in this way, only the ***number*** of overloaded implementations for the function of that row is shown.

Placeholder	Argument types	Description
`T`		a container or interval type
`P`		polymorphical container argument type
`J`		polymorphical iterator type
`K`		polymorphical element_iterator type for interval containers
`V`		various types `V`, that do dot fall in the categories above
1,2,...		number of implementations for this function
A		implementation generated by compilers
e	T::element_type	the element type of `interval_sets` or `std::sets`
i	T::segment_type	the segment type of of `interval_sets`
s	element sets	`std::set` or other models of the icl's set concept
S	interval_sets	one of the interval set types
b	T::element_type	type of `interval_map's` or `icl::map's` element value pairs
p	T::segment_type	type of `interval_map's` interval value pairs
m	element maps	`icl::map` icl's map type
M	interval_maps	one of the interval map types
d	discrete types	types with a least steppable discrete unit: Integral types, date/time types etc.
c	continuous types	types with (theoretically) infinitely many elements beween two values.

Table 13. Synopsis Functions and Overloads

T	intervals	interval sets	interval maps	element sets	element maps
Construct, copy, destruct					
`T::T()`	1	1	1	1	1
`T::T(const P&)`	A	**e i S**	**b p M**	1	1
`T& T::operator=(const P&)`	A	**S**	**M**	1	1
`void T::swap(T&)`		1	1	1	1
Containedness	intervals	interval sets	interval maps	element sets	element maps
`bool T::empty()const`		1	1	1	1
`bool is_empty(const T&)`	1	1	1	1	1
`bool contains(const T&, const P&)` `bool within(const P&, const T&)`	**e i**	**e i S**	**e i S b p M**	**e s**	**b m**
Equivalences and Orderings	intervals	interval sets	interval maps	element sets	element maps
`bool operator == (const T&, const T&)`	1	1	1	1	1
`bool operator != (const T&, const T&)`	1	1	1	1	1
`bool operator < (const T&, const T&)`	1	1	1	1	1
`bool operator > (const T&, const T&)`	1	1	1	1	1
`bool operator <= (const T&, const T&)`	1	1	1	1	1

T	intervals	interval sets	interval maps	element sets	element maps
`bool operator >= (const T&, const T&)`	1	1	1	1	1
`bool is_element_equal(const T&, const P&)`		**S**	**M**	1	1
`bool is_element_less(const T&, const P&)`		**S**	**M**	1	1
`bool is_element_greater(const T&, const P&)`		**S**	**M**	1	1
`bool is_distinct_equal(const T&, const P&)`			**M**		1
Size	intervals	interval sets	interval maps	element sets	element maps
`size_type T::size()const`		1	1	1	1
`size_type size(const T&)`	1	1	1	1	1
`size_type cardinality(const T&)`	1	1	1	1	1
`difference_type length(const T&)`	1	1	1		
`size_type iterative_size(const T&)`		1	1	1	1
`size_type interval_count(const T&)`		1	1		
Selection					
`J T::find(const P&)`		**e i**	**e i**	2	2

T	intervals	interval sets	interval maps	element sets	element maps
`J find(T&, const P&)`		e i	e i		
`codomain_type& operator[] (const domain_type&)`					1
`codomain_type operator() (const domain_type&)const`			1		1
Range					
`interval_type hull(const T&)`		1	1		
`T hull(const T&, const T&)`	1				
`domain_type lower(const T&)`	1	1	1		
`domain_type upper(const T&)`	1	1	1		
`domain_type first(const T&)`	1	1	1		
`domain_type last(const T&)`	1	1	1		
Addition	intervals	interval sets	interval maps	element sets	element maps
`T& T::add(const P&)`		**e i**	**b p**		**b**
`T& add(T&, const P&)`		**e i**	**b p**	**e**	**b**
`T& T::add(J pos, const P&)`		**i**	**p**		**b**
`T& add(T&, J pos, const P&)`		**i**	**p**	**e**	**b**
`T& operator +=(T&, const P&)`		**e i S**	**b p M**	**e s**	**b m**

T	intervals	interval sets	interval maps	element sets	element maps
`T operator +(T, const P&)` `T operator +(const P&, T)`		e i S	b p M	e s	b m
`T& operator |=( T&, const P&)`		e i S	b p M	e s	b m
`T operator |(T, const P&)` `T operator |(const P&, T)`		e i S	b p M	e s	b m
Subtraction					
`T& T::subtract(const P&)`		e i	b p		b
`T& subtract(T&, const P&)`		e i	b p	e	b
`T& operator -=(T&, const P&)`		e i S	e i S b p M	e s	b m
`T operator -(T, const P&)`		e i S	e i S b p M	e s	b m
`T left_subtract(T, const T&)`	1				
`T right_subtract(T, const T&)`	1				
Insertion	intervals	interval sets	interval maps	element sets	element maps
`V T::insert(const P&)`		e i	b p	e	b
`V insert(T&, const P&)`		e i	b p	e	b
`V T::insert(J pos, const P&)`		i	p	e	b
`V insert(T&, J pos, const P&)`		i	p	e	b

T	**intervals**	**interval sets**	**interval maps**	**element sets**	**element maps**
`T& insert(T&, const P&)`		e i S	b p M	e s	b m
`T & T::set(const P&)`			b p		1
`T& set_at(T&, const P&)`			b p		1
Erasure					
`void T::clear()`		1	1	1	1
`void clear(const T&)`		1	1	1	1
`T & T::erase(const P&)`		e i	e i b p	e	b p
`T& erase(T&, const P&)`		e i S	e i S b p M	e s	b m
`void T::erase(iterator)`		1	1	1	1
`void T::erase(iterator,iterator)`		1	1	1	1
Intersection	intervals	interval sets	interval maps	element sets	element maps
`void add_intersection(T&, const T&, const P&)`		e i S	e i S b p M		
`T& operator &=(T&, const P&)`		e i S	e i S b p M	e s	b m
`T operator & (T, const P&)` `T operator & (const P&, T)`	i	e i S	e i S b p M	e s	b m

T	intervals	interval sets	interval maps	element sets	element maps
`bool intersects(const T&, const P&)` `bool disjoint(const T&, const P&)`	**i**	**e i S**	**e i S b p M**	**e s**	**b m**
Symmetric difference					
`T & T::flip(const P&)`		**e i**	**b p**		**b**
`T& flip(T&, const P&)`		**e i**	**b p**	**e**	**b**
`T& operator ^=(T&, const P&)`		**e i S**	**b p M**	**e s**	**b m**
`T operator ^ (T, const P&)` `T operator ^ (const P&, T)`		**e i S**	**b p M**	**e s**	**b m**
Iteration	intervals	interval sets	interval maps	element sets	element maps
`J T::begin()`		2	2	2	2
`J T::end()`		2	2	2	2
`J T::rbegin()`		2	2	2	2
`J T::rend()`		2	2	2	2
`J T::lower_bound(const key_type&)`		2	2	2	2
`J T::upper_bound(const key_type&)`		2	2	2	2
`pair<J,J> T::equal_range(const key_type&)`		2	2	2	2
Element iteration	intervals	interval sets	interval maps	element sets	element maps
`K elements_begin(T&)`		2	2		

T	intervals	interval sets	interval maps	element sets	element maps
`K elements_end(T&)`		2	2		
`K elements_rbegin(T&)`		2	2		
`K elements_rend(T&)`		2	2		
Streaming, conversion	intervals	interval sets	interval maps	element sets	element maps
`std::basic_ostream operator << (basic_ostream&, const T&)`	1	1	1	1	1

Many but not all functions of **icl** intervals are listed in the table above. Some specific functions are summarized in the next table. For the group of the constructing functions, placeholders **d** denote discrete domain types and **c** denote continuous domain types `T::domain_type` for an interval_type `T` and an argument types `P`.

Table 14. Additional interval functions

T	discrete _interval	continuous _interval	right_open _interval	left_open _interval	closed _interval	open _interval
Interval bounds	dynamic	dynamic	static	static	static	static
Form			asymmetric	asymmetric	symmetric	symmetric
Construction						
`T singleton(const P&)`	d	c	d	d	d	d
`T construct(const P&, const P&)`	d	c	d c	d c	d	d
`T construct(const P&, const P&, interval_bounds)`	d	c				
`T hull(const P&, const P&)`	d	c	d c	d c	d	d
`T span(const P&, const P&)`	d	c	d c	d c	d	d
`static T right_open(const P&, const P&)`	d	c				
`static T left_open(const P&, const P&)`	d	c				
`static T closed(const P&, const P&)`	d	c				
`static T open(const P&, const P&)`	d	c				
Orderings						

T	discrete _interval	continuous _interval	right_open _interval	left_open _interval	closed _interval	open _interval
`bool exclusive_less(const T&, const T&)`	1	1	1	1	1	1
`bool lower_less(const T&, const T&)` `bool lower_equal(const T&, const T&)` `bool lower_less_equal(const T&, const T&)`	1	1	1	1	1	1
`bool upper_less(const T&, const T&)` `bool upper_equal(const T&, const T&)` `bool upper_less_equal(const T&, const T&)`	1	1	1	1	1	1
Miscellaneous						
`bool touches(const T&, const T&)`	1	1	1	1	1	1
`T inner_complement(const T&, const T&)`	1	1	1	1	1	1
`difference_type distance(const T&, const T&)`	1	1	1	1	1	1

Element iterators for interval containers

Iterators on **interval conainers** that are refered to in section *Iteration* of the function synopsis table are ***segment iterators***. They reveal the more implementation specific aspect, that the fundamental aspect abstracts from. Iteration over segments is fast, compared to an iteration over elements, particularly if intervals are large. But if we want to view our interval containers as containers of elements that are usable with std::algoritms, we need to iterate over elements.

Iteration over elements . . .

- is possible only for integral or discrete `domain_types`
- can be very ***slow*** if the intervals are very large.
- and is therefore ***depreciated***

On the other hand, sometimes iteration over interval containers on the element level might be desired, if you have some interface that works for `std::SortedAssociativeContainers` of elements and you need to quickly use it with an interval container. Accepting the poorer performance might be less bothersome at times than adjusting your whole interface for segment iteration.

Caution

So we advice you to choose element iteration over interval containers ***judiciously***. Do not use element iteration ***by default or habitual***. Always try to achieve results using namespace global functions or operators (preferably inplace versions) or iteration over segments first.

Customization

Intervals

The **icl** provides the possibility of customizing user defined interval class templates and class types with static interval borders to be used with interval containers.

There is a template `interval_traits`, that has to be instatiated for the user defined interval type, in order to provide associated types and basic functions. Bound types of the interval are assigned by specializing the template `interval_bound_type`.

Customize	Name	Description
associated types	`interval_type`	interval type of the partial specialisation for the user defined type
	`domain_type`	the domain or element type of the interval
	`domain_compare`	the ordering on the elements
basic functions	`construct(const domain_type&, const domain_type&)`	construct an interval
	`lower(const interval_type&)`	select the interval's lower bound
	`upper(const interval_type&)`	select the interval's upper bound
interval bounds	`interval_bound_type<interval_type>{...}`	specialize meta function `interval_bound_type` to assign one of the 4 bound types to the user defined interval.

How to do the customization in detail is shown in example custom interval.

Implementation

The previous section gave an overview over the interface of the **icl** outlining class templates, associated types and polymorphic functions and operators. In preparation to the next section, that describes the **icl's** polymorphic functions in more detail together with ***complexity characteristics***, this section summarizes some general information on implementation and performance.

STL based implementation

The **implementation** of the **icl's** containers is based on **std::set** and **std::map**. So the underlying data structure of interval containers is a red black tree of intervals or interval value pairs. The element containers `std::set` and `icl::map` are wrapper classes of `std::set` and `std::map`. Interval containers are then using `std::sets` of intervals or `icl::maps` of interval value pairs as implementing containers. So all the ***complexity characteristics*** of icl containers are based on and limited by the ***red-black tree implementation*** of the underlying std::AssociativeContainers.

Iterative size

Throughout the documentation on complexity, big *O* expressions like *O(n)* or *O(m log n)* refer to container sizes *n* and *m*. In this documentation these sizes ***do not*** denote to the familiar `size` function that returns the ***number of elements*** of a container. Because for an interval container

```
interval_set<int> mono;
mono += interval<int>::closed(1,5); // {[1 ... 5]}
mono.size()             == 5;         // true, 5 elements
mono.interval_count() == 1;         // true, only one interval
```

it's size and the number of contained intervals is usually different. To refer uniformly to a *size* that matters for iteration, which is the decisive kind of size concerning algorithmic behavior there is a function

```
bool T::iterative_size()const; // Number of entities that can be iterated over.
```

for all element and interval containers of the icl. So for complexity statements throughout the icl's documentation the sizes will be `iterative_sizes` and big *O* expressions like *O(m log n)* will refer to sizes

```
n = y.iterative_size();
m = x.iterative_size();
```

for containers `y` and `x`. Note that

```
iterative_size
```

refers to the primary entities, that we can iterate over. For interval containers these are intervals or segments.

```
Itervative_size
```

never refers to element iteration for interval containers.

Complexity

Complexity of element containers

Since *element containers* `std::set` and `icl::map` are only extensions of stl::set and stl::map, their complexity characteristics are accordingly. So their major operations insertion (addition), deletion and search are all using logarithmic time.

Complexity of interval containers

The operations on *interval containers* behave differently due to the fact that intervals unlike elements can overlap any number of other intervals in a container. As long as intervals are relatively small or just singleton, interval containers behave like containers of elements. For large intervals however time consumption of operations on interval containers may be worse, because most or all intervals of a container may have to be visited. As an example, time complexity of ***Addition*** on interval containers is briefly discussed.

More information on ***complexity characteristics*** of **icl's** functions is contained in section Function Reference

Time Complexity of Addition

The next table gives the time complexities for the overloaded `operator +=` on interval containers. The instance types of `T` are given as column headers. Instances of type parameter `P` are denoted in the second column. The third column contains the specific kind of complexity statement. If column three is empty ***worst case*** complexity is given in the related row.

Table 15. Time Complexity of Addition:

	P		**interval set**	**separate interval set**	**split interval set**	**interval map**	**split interval map**
`T& operator +=(T& object, const P& addend)`	`T::element_type`		*O(log n)*	*O(log n)*	*O(log n)*	*O(log n)*	*O(log n)*
	`T::segment_type`	best case	*O(log n)*	*O(log n)*	*O(log n)*	*O(log n)*	*O(log n)*
		worst case	*O(n)*	*O(n)*	*O(n)*	*O(n)*	*O(n)*
		amortized	*O(log n)*	*O(log n)*			
	`interval_sets`		*O(m log(n+m))*	*O(m log(n+m))*	*O(m log(n+m))*		
	`interval_maps`					*O(m log(n+m))*	*O(m log(n+m))*

Adding an *element* or *element value pair* is always done in *logarithmic time*, where *n* is the number of intervals in the interval container. The same row of complexities applies to the insertion of a *segment* (an interval or an interval value pair) in the ***best case***, where the inserted segment does overlap with only a ***small*** number of intervals in the container.

In the ***worst case***, where the inserted segment overlaps with all intervals in the container, the algorithms iterate over all the overlapped segments. Using inplace manipulations of segments and hinted inserts, it is possible to perform all necessary operations on each iteration step in *constant time*. This results in ***linear worst case time*** complexity for segment addition for all interval containers.

After performing a worst case addition for an `interval_set` or a `separate_interval_sets` adding an interval that overlaps *n* intervals, we need *n* non overlapping additions of *logarithmic time* before we can launch another *O(n)* worst case addition. So we have only a ***logarithmic amortized time*** for the addition of an interval or interval value pair.

For the addition of ***interval containers*** complexity is *O(m log(n+m))*. So for the ***worst case***, where the container sizes *n* and *m* are equal and both containers cover the same ranges, the complexity of container addition is ***loglinear***. For other cases, that occur frequently in real world applications performance can be much better. If the added container `operand` is much smaller than `object` and the intervals in `operand` are relatively small, performance can be *logarithmic*. If *m* is small compared with *n* and intervals in `operand` are large, performance tends to be *linear*.

Inplace and infix operators

For the major operations *addition, subtraction, intersection* of **icl** containers and for *symmetric difference* inplace `operators += |=, -=, &=` and `^=` are provided.

For every ***inplace*** operator

```
T& operator o= (T& object, const P& operand)
```

the **icl** provides corresponding ***infix*** operators.

```
T operator o (T object, const P& operand){ return object o= operand; }
T operator o (const P& operand, T object){ return object o= operand; }
```

From this implementation of the infix `operator o` the compiler will hopefully use return value optimization (RVO) creating no temporary object and performing one copy of the first argument `object`.

Caution

Compared to the *inplace* `operator o=` every use of an *infix* `operator o` requires ***one extra copy*** of the first argument `object` that passes a container.

Use infix operators only, if

- efficiency is not crucial, e.g. the containers copied are small.
- a concise and short notation is more important than efficiency in your context.
- you need the result of operator `o=` as a copy anyway.

Time Complexity of infix `operators o`

The time complexity of all infix operators of the **icl** is biased by the extra copy of the `object` argument. So all infix `operators o` are at least *linear* in `n = object.iterative_size()`. Taking this into account, the complexities of all infix operators can be determined from the corresponding inplace `operators o=` they depend on.

Function Reference

Section Function Synopsis above gave an overview of the polymorphic functions of the icl. This is what you will need to find the desired possibilities to combine icl functions and objects most of the time. The functions and overloads that you intuitively expect should be provided, so you won't need to refer to the documentation very often.

If you are interested

- in the ***specific design of the function overloads***,
- in ***complexity characteristics*** for certain overloads
- or if the compiler ***refuses to resolve*** specific function application you want to use,

refer to this section that describes the polymorphic function families of the icl in detail.

Placeholders

For a concise representation the same placeholders will be used that have been introduced in section Function Synopsis.

More specific function documentation

This section covers the most important polymorphical and namespace global functions of the **icl**. More specific functions can be looked up in the doxygen generated reference documentation.

Overload tables

Many of the **icl's** functions are overloaded for elements, segments, element and interval containers. But not all type combinations are provided. Also the admissible type combinations are different for different functions and operations. To concisely represent the overloads that can be used we use synoptical tables that contain possible type combinations for an operation. These are called ***overload tables***. As an example the overload tables for the inplace intersection `operator &=` are given:

```
// overload tables for
T& operator &= (T&, const P&)

element containers:     interval containers:
&= | e b s m            &= | e i b p S M
---+--------            ---+------------
s  | s   s              S  | S S     S
m  | m m m m            M  | M M M M M M
```

For the binary `T& operator &= (T&, const P&)` there are two different tables for the overloads of element and interval containers. The first argument type `T` is displayed as row headers of the tables. The second argument type `P` is displayed as column headers of the tables. If a combination of `T` and `P` is admissible the related cell of the table is non empty. It displays the result type of the operation. In this example the result type is always equal to the first argument.

The possible types that can be instantiated for `T` and `P` are element, interval and container types abbreviated by placeholders that are defined here and can be summarized as

s : element set, **S** : interval sets, **e** : elements, **i** : intervals
m:element map, **M**:interval maps, **b**:element-value pairs, **p**:interval-value pairs

Segmentational Fineness

For overloading tables on infix operators, we need to break down `interval_sets` and `interval_maps` to the specific class templates

S1	interval_set	**S2**	separate_interval_set	**S3**	split_interval_set
M1	interval_map			**M3**	split_interval_map

choosing **Si** and **Mi** as placeholders.

The indices **i** of **Si** and **Mi** represent a property called ***segmentational fineness*** or short ***fineness***, which is a ***type trait*** on interval containers.

`segmentational_fineness<`**Si**`>::value` == **i**
`segmentational_fineness<`**Mi**`>::value` == **i**

Segmentational fineness represents the fact, that for interval containers holding the same elements, a splitting interval container may contain more segments as a separating container which in turn may contain more segments than a joining one. So for an

```
operator >
```

where

```
x > y   // means that
x is_finer_than y

// we have

finer                                             coarser
split_interval_set                           interval_set
                     > separate_interval_set >
split_interval_map                           interval_map
```

This relation is needed to resolve the instantiation of infix operators e.g. `T operator + (P, Q)` for two interval container types `P` and `Q`. If both `P` and `Q` are candidates for the result type `T`, one of them must be chosen by the compiler. We choose the type that is segmentational finer as result type `T`. This way we do not loose the ***segment information*** that is stored in the ***finer*** one of the container types `P` and `Q`.

```
// overload tables for
T operator + (T, const P&)
T operator + (const P&, T)

element containers:     interval containers:
+   | e b s m           +  | e  i  b  p  S1 S2 S3 M1 M3
---+--------            ---+--------------------------
e   |     s             e  |             S1 S2 S3
b   |       m           i  |             S1 S2 S3
s   | s   s             b  |                      M1 M3
m   |   m   m           p  |                      M1 M3
                        S1 | S1 S1       S1 S2 S3
                        S2 | S2 S2       S2 S2 S3
                        S3 | S3 S3       S3 S3 S3
                        M1 |       M1 M1          M1 M3
                        M3 |       M3 M3          M3 M3
```

So looking up a type combination for e.g. `T operator + (interval_map, split_interval_map)` which is equivalent to `T operator + (M1, M3)` we find for row type `M1` and column type `M3` that `M3` will be assigned as result type, because `M3` is finer than `M1`. So this type combination will result in choosing this

```
split_interval_map operator + (const interval_map&, split_interval_map)
```

implementation by the compiler.

Key Types

In an **stl** map `map<K,D>` the first parameter type of the map template `K` is called `key_type`. It allows to select key-value pairs pairs via `find(const K&)` and to remove key-value pairs using `erase(const K&)`. For icl Maps we have generalized key types to a larger set of types. Not only the `key_type` (`domain_type`) but also an interval type and a set type can be ***key types***, that allow for ***selection*** and ***removal*** of map elements segments and submaps.

Table 16. Selection of elements, segments and sub maps using key types

	M: `interval_maps`	**m**: **icl_map**
e: `domain_type`	key value pair	key value pair
i: `interval_type`	interval value pair	
S: `interval_sets`	interval map	
s: `std::set`		interval map

Subtraction, ***erasure***, ***intersection*** and ***containedness*** predicates can be used with those kinds of key types. For instance, the overload table for intersection

```
// overload tables for
T& operator &= (T&, const P&)

element containers:     interval containers:
&= | e b s m            &= | e i b p S M
---+--------            ---+------------
s  | s   s              S  | S S     S
m  | m m m m            M  | M M M M M M
```

has a part that that allows for selection by key objects

```
element containers:     interval containers:
&= | e b s m            &= | e i b p S M
---+--------            ---+------------
s  | s   s              S  | S S     S
m  | m   m              M  | M M     M
```

and another part that provides overloads for generalized intersection:

```
element containers:     interval containers:
&= | e b s m            &= | e i b p S M
---+--------            ---+------------
s  | s   s              S  | S S     S
m  |   m   m            M  |     M M   M
```

For `Sets`, the ***key types*** defined for maps are identical with the set types themselves. So the distinction between the function groups ***selection by key*** and ***generalized intersection*** fall together in the well known ***set intersection***.

Construct, copy, destruct

Construct, copy, destruct	intervals	interval sets	interval maps	element sets	element maps
`T::T()`	1	1	1	1	1
`T::T(const P&)`	A	**e i S**	**b p M**	1	1
`T& T::operator=(const P&)`	A	**S**	**M**	1	1
`void T::swap(T&)`		1	1	1	1

All **icl** types are ***regular types***. They are ***default constructible***, ***copy constructible*** and ***assignable***. On icl Sets and Maps a `swap` function is available, that allows for **constant time** swapping of container contents. The *regular and swappable part* of the basic functions and their complexities are described in the tables below.

Regular and swap	intervals	interval sets	interval maps	element sets	element maps
`T::T()`	*O(1)*	*O(1)*	*O(1)*	*O(1)*	*O(1)*
`T::T(const T&)`	*O(1)*	*O(n)*	*O(n)*	*O(n)*	*O(n)*
`T& T::operator=(const T&)`	*O(1)*	*O(n)*	*O(n)*	*O(n)*	*O(n)*
`void T::swap(T&)`		*O(1)*	*O(1)*	*O(1)*	*O(1)*

where *n* = `iterative_size(x)`.

Construct, copy, destruct	Description
`T::T()`	Object of type T is default constructed.
`T::T(const T& src)`	Object of type T is copy constructed from object `src`.
`T& T::operator=(const T& src)`	Assigns the contents of src to `*this` object. Returns a reference to the assigned object.
`void T::swap(T& src)`	Swaps the content containers `*this` and `src` in constant time.

In addition we have overloads of constructors and assignment operators for icl container types.

```
// overload tables for constructors
T::T(const P& src)

element containers:     interval containers:
T \ P | e b s m         T \ P | e i b p S M
------+--------         ------+------------
s     | s   s           S     | S S     S
m     |   m   m         M     |     M M   M
```

For an object `dst` of type `T` and an argument `src` of type `P` let

```
n = iterative_size(dst);
m = iterative_size(src);
```

in the following tables.

Table 17. Time Complexity for overloaded constructors on element containers

`T(const P& src)`	**domain type**	**domain mapping type**	**interval sets**	**interval maps**
`std::set`	*O(log n)*		*O(m)*	
`icl::map`		*O(log n)*		*O(m)*

Time complexity characteristics of inplace insertion for interval containers is given by this table.

Table 18. Time Complexity for overloaded constructors on interval containers

`T(const P& src)`	**domain type**	**interval type**	**domain mapping type**	**interval mapping type**	**interval sets**	**interval maps**
interval_sets	*O(1)*	*O(1)*			*O(m)*	
interval_maps			*O(1)*	*O(1)*		*O(m)*

```
// overload tables for assignment
T& operator = (const P& src)

interval containers:
T \ P | S M
------+----
S     | S
M     |   M
```

The assignment `T& operator = (const P& src)` is overloaded within interval containers. For all type combinations we have ***linear time complexity*** in the maximum of the `iterative_size` of `dst` and `src`.

Back to section . . .

Function Synopsis
Interface

Containedness

Containedness	intervals	interval sets	interval maps	element sets	element maps
`bool T::empty()const`		1	1	1	1
`bool is_empty(const T&)`	1	1	1	1	1
`bool contains(const T&, const P&)` `bool within(const P&, const T&)`	**e i**	**e i S**	**e i S b p M**	**e s**	**b m**

This group of functions refers to ***contain***edness which should be fundamental to ***contain***ers. The function `contains` is overloaded. It covers different kinds of containedness: Containedness of elements, segments, and sub containers.

Containedness	*O(...)*	Description
`bool T::empty()const` `bool is_empty(const T&)`	*O(1)*	Returns `true`, if the container is empty, `false` otherwise.
`bool contains(const T&, const P&)` `bool within(const P&, const T&)`	*see below*	Returns `true`, if `super` container contains object `sub`.
	where	*n* `= iterative_size(sub)`
		m `= iterative_size(super)`

```
// overload tables for
bool contains(const T& super, const P& sub)
bool   within(const P& sub, const T& super)

element containers:   interval containers:
T\P| e b s m          T\P| e i b p S M
--------+---          --------+-------
 s | 1   1             S | 1 1     1
 m | 1 1 1 1           M | 1 1 1 1 1 1
```

The overloads of `bool contains(const T& super, const P& sup)` cover various kinds of containedness. We can group them into a part (1) that checks if an element, a segment or a container *of same kinds* is contained in an element or interval container

```
// (1) containedness of elements, segments or containers of same kind
T\P| e b s m          T\P| e i b p S M
---+--------          ---+------------
 s | 1   1             S | 1 1     1
 m |   1   1           M |     1 1   1
```

and another part (2) that checks the containedness of *key objects*, which can be *elements* an *intervals* or a *sets*.

```
// (2) containedness of key objects.
T\P| e b s m          T\P| e i b p S M
---+--------          ---+------------
 s | 1   1             S | 1 1     1
 m | 1   1             M | 1 1     1
```

For type **m** = `icl::map`, a key element (**m**`::domain_type`) and an `std::set` (**m**`::set_type`) can be a ***key object***.

For an interval map type **M**, a key element (**M**`::domain_type`), an interval (**M**`::interval_type`) and an ***interval set***, can be ***key objects***.

Complexity characteristics for function `bool contains(const T& super, const P& sub)const` are given by the next tables where

```
n = iterative_size(super);
m = iterative_size(sub); //if P is a container type
```

Table 19. Time Complexity for function contains on element containers

`bool contains(const T& super, const P& sub)` **`bool within(const P& sub, const T& super)`**	**domain type**	**domain mapping type**	**std::set**	**icl::map**
`std::set`	*O(log n)*		*O(m log n)*	
`icl::map`	*O(log n)*	*O(log n)*	*O(m log n)*	*O(m log n)*

Table 20. Time Complexity for functions contains and within on interval containers

`bool contains(const T& super, const P& sub)` `bool within(const P& sub, const T& super)`		**domain type**	**interval type**	**domain mapping type**	**interval mapping type**	**interval sets**	**interval maps**
interval_sets	`interval_set`	*O(log n)*	*O(log n)*			*O(m log n)*	
	`separate_interval_set` `split_interval_set`	*O(log n)*	*O(n)*			*O(m log n)*	
interval_maps	`interval_map`	*O(log n)*	*O(log n)*	*O(log n)*	*O(log n)*	*O(m log n)*	*O(m log n)*
	`split_interval_map`	*O(log n)*	*O(n)*	*O(log n)*	*O(n)*	*O(m log n)*	*O(m log n)*

All overloads of containedness of containers in containers

```
bool contains(const T& super, const P& sub)
bool   within(const P& sub, const T& super)
```

are of ***loglinear*** time: *O(m log n)*. If both containers have same iterative_sizes so that *m = n* we have the worst case (*O(n log n)*). There is an alternative implementation that has a ***linear*** complexity of *O(n+m)*. The loglinear implementation has been chosen, because it can be faster, if the container argument is small. In this case the loglinear implementation approaches logarithmic behavior, whereas the linear implementation stays linear.

Back to section . . .

Function Synopsis
Interface

Equivalences and Orderings

Synopsis

Equivalences and Orderings	intervals	interval sets	interval maps	element sets	element maps
Segment Ordering					
`bool operator == (const T&, const T&)`	1	1	1	1	1
`bool operator != (const T&, const T&)`	1	1	1	1	1
`bool operator < (const T&, const T&)`	1	1	1	1	1
`bool operator > (const T&, const T&)`	1	1	1	1	1
`bool operator <= (const T&, const T&)`	1	1	1	1	1
`bool operator >= (const T&, const T&)`	1	1	1	1	1
Element Ordering					
`bool is_element_equal(const T&, const P&)`		**S**	**M**	1	1
`bool is_element_less(const T&, const P&)`		**S**	**M**	1	1
`bool is_element_greater(const T&, const P&)`		**S**	**M**	1	1
Distinct Equality					
`bool is_distinct_equal(const T&, const P&)`			**M**		1

Less on Intervals

	Types	
`x < y`	**i**	`x` begins before `y` or, for equal beginnings `x` ends before `y`

Lexicographical Ordering

All common equality and compare operators are defined for all objects of the **icl**. For all **icl** containers equality and compare operators implement lexicographical equality and lexicographical comparison, that depends on the equality of template parameter `Compare`. This includes the less ordering on intervals, that can be perceived as the sequence of elements between their lower and upper bound. This generalized lexicogrphical comparison in intervals can also be specified this way:

`x < y`	`:=`	`x` begins before `y` or, for equal beginnings `x` ends before `y`.
		The other operators can be deduced in the usual way
`x > y`	`:=`	`y < x`
`x <= y`	`:=`	`!(y < x)`
`x >= y`	`:=`	`!(x < y)`
`x == y`	`:=`	`!(x < y) && !(y < x)` induced equivalence
`x != y`	`:=`	`!(x == y)`

Equality and compare operators are defined for all **icl** objects but there are no overloads between different types.

Containers of different segmentation are different, even if their elements are the same:

```
split_interval_set<time> w1, w2; //Pseudocode
w1 = {[Mon       ..       Sun)}; //split_interval_set containing a week
w2 = {[Mon .. Fri)[Sat .. Sun)}; //Same week split in work and week end parts.
w1 == w2;                        //false: Different segmentation
is_element_equal(w1,w2);         //true:  Same elements contained
```

Complexity is ***linear*** in the `iterative_size` of the shorter container to compare.

Sequential Element Ordering

The ***Sequential Element Ordering*** abstracts from the way in which elements of interval containers are clustered into intervals: it's ***segmentation***.

So these equality and compare operations can be applied within interval container types. The admissible type combinations are summarized in the next overload table.

```
// overload tables for
bool is_element_equal  (const T&, const P&)
bool is_element_less   (const T&, const P&)
bool is_element_greater(const T&, const P&)

element containers:     interval containers:
T\P| s m                T\P| S1 S2 S3 M1 M3
---+----                ---+---------------
s  | 1                  S1 | 1  1  1
m  |   1                S2 | 1  1  1
                        S3 | 1  1  1
                        M1 |          1  1
                        M3 |          1  1
```

For element containers lexicographical equality and sequential element equality are identical.

The **complexity** of sequential element comparison functions is ***linear*** in the `iterative_size` of the larger container.

Distinct Equality

Distinct Equality is an equality predicate that is available for `icl::maps` and `interval_maps`. It yields true, if two maps are sequential element equal except for value pairs whose associated values are identity elements.

Complexity is linear in the `iterative_size` of the larger container to compare.

See also . . .

Semantics

Back to section . . .

Function Synopsis
Interface

Size

Size	intervals	interval sets	interval maps	element sets	element maps
`size_type T::size()const` `size_type size(const T&)`	*O(1)*	*O(n)*	*O(n)*	*O(1)*	*O(1)*
`size_type cardinality(const T&)`	*O(1)*	*O(n)*	*O(n)*	*O(1)*	*O(1)*
`difference_type length(const T&)`	*O(1)*	*O(n)*	*O(n)*		
`size_type iterative_size(const T&)`		*O(1)*	*O(1)*	*O(1)*	*O(1)*
`size_type interval_count(const T&)`		*O(1)*	*O(1)*		

For **icl** containers the single `size` function known from std containers branches into tree to five different members functions. The table above shows the types, `size` functions are implemented for, together with their **complexities**. Linear complexities *O(n)* refer to the container's `iterative_size`:

```
n = y.iterative_size()
```

The next table gives a short definition for the different size functions.

Size	Types	Description
`size_type interval_count(const T&)`	**S M**	The number of intervals of an interval container.
`size_type iterative_size(const T&)`	**S M s m**	The number of objects in an icl container that can be iterated over.
`difference_type length(const T&)`	**i S M**	The length of an interval or the sum of lengths of an interval container's intervals, that's `domain_type` has a `difference_type`.
`size_type cardinality(const T&)`	**i S M s m**	The number of elements of an interval or a container. For continuous data types cardinality can be *infinite*.
`size_type T::size()const` `size_type size(const T&)`	**i S M s m**	The number of elements of an interval or a container, which is also it's `cardinality`.

Back to section . . .

Function Synopsis
Interface

Range

Range	intervals	interval sets	interval maps	condition
`interval_type hull(const T&)`		*O(1)*	*O(1)*	
`T hull(const T&, const T&)`	*O(1)*			
`domain_type lower(const T&)`	*O(1)*	*O(1)*	*O(1)*	
`domain_type upper(const T&)`	*O(1)*	*O(1)*	*O(1)*	
`domain_type first(const T&)`	*O(1)*	*O(1)*	*O(1)*	`is_discrete<domain_type>::value`
`domain_type last(const T&)`	*O(1)*	*O(1)*	*O(1)*	`is_discrete<domain_type>::value`

The table above shows the availability of functions `hull`, `lower`, `upper`, `first` and `last` on intervals and interval containers that are all of ***constant time complexity***. Find the functions description and some simple properties below.

Range	Types	Description
`interval_type hull(const T&)`	**S M**	`hull(x)` returns the smallest interval that contains all intervals of an interval container `x`.
`T hull(const T&, const T&)`	**S M**	`hull(i,j)` returns the smallest interval that contains intervals `i` abd 'j'.
`domain_type lower(const T&)`	**i S M**	`lower(x)` returns the lower bound of an interval or interval container `x`.
`domain_type upper(const T&)`	**i S M**	`upper(x)` returns the upper bound of an interval or interval container `x`.
`domain_type first(const T&)`	**i S M**	`first(x)` returns the first element of an interval or interval container `x`. `first(const T&)` is defined for a discrete `domain_type` only.
`domain_type last(const T&)`	**i S M**	`last(x)` returns the last element of an interval or interval container `x`. `last(const T&)` is defined for a discrete `domain_type` only.

```
// for interval_containers x:
lower(hull(x)) == lower(x)
upper(hull(x)) == upper(x)
first(hull(x)) == first(x)
last(hull(x))  == last(x)
```

Back to section . . .

Function Synopsis
Interface

Selection

Selection	**interval sets**	**interval maps**	**element sets**	**element maps**	**condition**
`iterator T::find(const domain_type&)`			*O(log n)*	*O(log n)*	
`const_iterator T::find(const domain_type&)const`	*O(log n)*	*O(log n)*	*O(log n)*	*O(log n)*	
`const_iterator find(T&, const domain_type&)`	*O(log n)*	*O(log n)*			
`const_iterator T::find(const interval_type&)const`	*O(log n)*	*O(log n)*			
`const_iterator find(T&, const interval_type&)`	*O(log n)*	*O(log n)*			
`codomain_type& operator[] (const domain_type&)`				*O(log n)*	
`codomain_type operator() (const domain_type&)const`		*O(log n)*		*O(log n)*	`is_total<T>::value`

- All time **complexities** are ***logarithmic*** in the containers `iterative_size()`.
- `operator()` is available for total maps only.
- `interval_type` is also the interval container's `key_type`

Selection	**Types**	**Description**
`iterator T::find(const domain_type& x)`	**s m**	Searches the container for the element `x` and return an iterator to it, if `x` is found. Otherwise `find` returns iterator `end()`.
`const_iterator T::find (const domain_type& x)const`	**s m**	Const version of `find` above.
`const_iterator T::find (const domain_type& x)const` `const_iterator find (T&, const domain_type& x)`	**S M**	For interval containers `c`, `c.find(x)` or `icl::find(c,x)` searches a key element `x` and returns an iterator to the interval containing the element `x`.
`const_iterator T::find (const interval_type& x)const` `const_iterator find (T&, const interval_type& x)`	**S M**	For interval containers `c`, `c.find(y)` or `icl::find(c,y)` searches an interval `y` and returns an iterator to the first interval in `c` that overlaps with `y`.
`codomain_type& operator[] (const domain_type& x)`	**m**	For the key element `x` the operator returns a reference to the mapped value. A pair `std::pair(x,codomain_type())` will be inserted, of `x` is not found in the map.
`codomain_type operator() (const domain_type& x)const`	**M m**	Returns the mapped value for a key `x`. The operator is only available for ***total*** maps.

See also . . .

Intersection

Back to section . . .

Function Synopsis
Interface

Addition

Synopsis

Addition	**interval sets**	**interval maps**	**element sets**	**element maps**
`T& T::add(const P&)`	**e i**	**b p**		**b**
`T& add(T&, const P&)`	**e i**	**b p**	**e**	**b**
`T& T::add(J pos, const P&)`	**i**	**p**		**b**
`T& add(T&, J pos, const P&)`	**i**	**p**	**e**	**b**
`T& operator +=(T&, const P&)`	**e i S**	**b p M**	**e s**	**b m**
`T operator + (T, const P&)` `T operator + (const P&, T)`	**e i S**	**b p M**	**e s**	**b m**
`T& operator |=( T&, const P&)`	**e i S**	**b p M**	**e s**	**b m**
`T operator | (T, const P&)` `T operator | (const P&, T)`	**e i S**	**b p M**	**e s**	**b m**

Functions and operators that implement ***Addition*** on **icl** objects are given in the table above. `operator |=` and `operator |` are behavioral identical to `operator +=` and `operator +`. This is a redundancy that has been introduced deliberately, because a *set union* semantics is often attached `operators |=` and `|`.

	Description of Addition
`Sets`	Addition on Sets implements ***set union***
`Maps`	Addition on Maps implements a ***map union*** function similar to *set union*. If, on insertion of an element value pair `(k,v)` it's key `k` is in the map already, the addition function is propagated to the associated value. This functionality has been introduced as *aggregate on collision* for element maps and *aggregate on overlap* for interval maps. Find more on *addability of maps* and related *semantic issues* following the links. Examples, demonstrating Addition on interval containers are *overlap counter*, *party* and *party's height average.*

For `Sets` ***addition*** and ***insertion*** are implemented identically. Functions `add` and `insert` collapse to the same function. For `Maps` ***addition*** and ***insertion*** work differently. Function `add` performs aggregations on collision or overlap, while function `insert` only inserts values that do not yet have key values.

Functions

The admissible combinations of types for member function `T& T::add(const P&)` can be summarized in the ***overload table*** below:

```
// overload table for    T\P| e i b p
T& T::add(const P&)      ---+--------
T& add(T&, const P&)      s | s
                          m |     m
                          S | S S
                          M |     M M
```

The next table contains complexity characteristics for `add`.

Table 21. Time Complexity for member function add on icl containers

`T& T::add(const P&)` `T& add(T&, const P&)`	domain type	interval type	domain mapping type	interval mapping type
`std::set`	*O(log n)*			
`icl::map`			*O(log n)*	
`interval_set` `separate_interval_set`	*O(log n)*	*amortized O(log n)*		
`split_interval_set`	*O(log n)*	*O(n)*		
`interval_map` `split_interval_map`			*O(log n)*	*O(n)*

Hinted addition

Function `T& T::add(T::iterator prior, const P& addend)` allows for an addition in ***constant time***, if `addend` can be inserted right after iterator `prior` without collision. If this is not possible the complexity characteristics are as stated for the non hinted addition above. Hinted addition is available for these combinations of types:

```
// overload table for addition with hint        T\P| e i b p
T& T::add(T::iterator prior, const P&)          ---+--------
T& add(T&, T::iterator prior, const P&)          s | s
                                                 m |     m
                                                 S |   S
                                                 M |       M
```

Inplace operators

The possible overloads of inplace `T& operator += (T&, const P&)` are given by two tables, that show admissible combinations of types. Row types show instantiations of argument type `T`. Columns types show show instantiations of argument type `P`. If a combination of argument types is possible, the related table cell contains the result type of the operation. Placeholders **e i b p s S m M** will be used to denote *elements, intervals, element value pairs, interval value pairs, element sets, interval sets, element maps* and *interval maps*. The first table shows the overloads of `+=` for *element containers* the second table refers to *interval containers*.

```
// overload tables for             element containers:     interval containers:
T& operator += (T&, const P&)      += | e b s m            += | e i b p S M
                                   ---+--------            ---+------------
                                   s  | s   s              S  | S S     S
                                   m  |   m   m            M  |     M M   M
```

For the definition of admissible overloads we separate *element containers* from *interval containers*. Within each group all combinations of types are supported for an operation, that are in line with the **icl's** design and the sets of laws, that establish the **icl's** semantics.

Overloads between *element containers* and *interval containers* could also be defined. But this has not been done for pragmatical reasons: Each additional combination of types for an operation enlarges the space of possible overloads. This makes the overload resolution by compilers more complex, error prone and slows down compilation speed. Error messages for unresolvable or ambiguous overloads are difficult to read and understand. Therefore overloading of namespace global functions in the **icl** are limited to a reasonable field of combinations, that are described here.

Complexity

For different combinations of argument types `T` and `P` different implementations of the `operator +=` are selected. These implementations show different complexity characteristics. If `T` is a container type, the combination of domain elements (**e**) or element value pairs (**b**) is faster than a combination of intervals (**i**) or interval value pairs (**p**) which in turn is faster than the combination of element or interval containers. The next table shows *time complexities* of addition for **icl's** element containers.

Sizes `n` and `m` are in the complexity statements are sizes of objects `T y` and `P x`:

```
n = iterative_size(y);
m = iterative_size(x); //if P is a container type
```

Note, that for an interval container the number of elements `T::size` is different from the number of intervals that you can iterate over. Therefore a function `T::iterative_size()` is used that provides the desired kind of size.

Table 22. Time Complexity for inplace Addition on element containers

`T& operator += (T& y, const P& x)`	domain type	domain mapping type	__ch_icl_sets__	__ch_icl_maps__
`std::set`	*O(log n)*		*O(m)*	
`icl::map`		*O(log n)*		*O(m)*

Time complexity characteristics of inplace addition for interval containers is given by this table.

Table 23. Time Complexity for inplace Addition on interval containers

`T& operator += (T& y, const P& x)`		domain type	interval type	domain mapping type	interval mapping type	interval sets	interval maps
interval_sets	`interval_set` `separate_interval_set`	*O(log n)*	*amortized O(log n)*			*O(m log(n+m))*	
	`split_interval_set`	*O(log n)*	*O(n)*			*O(m log(n+m))*	
interval_maps				*O(log n)*	*O(n)*		*O(m log(n+m))*

Since the implementation of element and interval containers is based on the link red-black tree implementation of std::AssociativeContainers, we have a logarithmic complexity for addition of elements. Addition of intervals or interval value pairs is amortized logarithmic for `interval_sets` and `separate_interval_sets` and linear for `split_interval_sets` and `interval_maps`. Addition is linear for element containers and loglinear for interval containers.

Infix operators

The admissible type combinations for infix `operator +` are defined by the overload tables below.

```
// overload tables for             element containers:     interval containers:
T operator + (T, const P&)         +  | e b s m            +  | e  i  b  p  S1 S2 S3 M1 M3
T operator + (const P&, T)         ---+--------            ---+--------------------------
                                   e  |     s              e  |             S1 S2 S3
                                   b  |       m            i  |             S1 S2 S3
                                   s  | s   s              b  |                      M1 M3
                                   m  |   m   m            p  |                      M1 M3
                                                           S1 | S1 S1       S1 S2 S3
                                                           S2 | S2 S2       S2 S2 S3
                                                           S3 | S3 S3       S3 S3 S3
                                                           M1 |       M1 M1          M1 M3
                                                           M3 |       M3 M3          M3 M3
```

See also . . .

Subtraction
Insertion

Back to section . . .

Function Synopsis
Interface

Subtraction

Synopsis

Subtraction	intervals	interval sets	interval maps	element sets	element maps
`T& T::subtract(const P&)`		e i	b p		b
`T& subtract(T&, const P&)`		e i	b p	e	b
`T& operator -=(T&, const P&)`		e i S	e i S b p M	e s	b m
`T operator -(T, const P&)`		e i S	e i S b p M	e s	b m
`T left_subtract(T, const T&)`	1				
`T right_subtract(T, const T&)`	1				

Functions and operators that implement ***Subtraction*** on **icl** objects are given in the table above.

	Description of Subtraction
`Sets`	Subtraction on Sets implements ***set difference***
`Maps`	Subtraction on Maps implements a ***map difference*** function similar to *set difference*. If, on subtraction of an element value pair `(k,v)` it's key `k` is in the map already, the subtraction function is propagated to the associated value. On the associated value an aggregation is performed, that reverses the effect of the corresponding addition function. Find more on *subtractability of maps* and related *semantic issues* following the links.

Functions

The admissible combinations of types for subtraction functions can be summarized in the ***overload table*** below:

```
// overload table for             T\P| e i b p
T& T::subtract(const P&)          ---+--------
T& subtract(T&, const P&)           s | s
                                    m |     m
                                    S | S S
                                    M |     M M
```

The next table contains complexity characteristics for `subtract`.

Table 24. Time Complexity for function subtract on icl containers

`T& T::subtract(const P&)` `T& subtract(T&, const P&)`	domain type	interval type	domain mapping type	interval mapping type
`std::set`	*O(log n)*			
`icl::map`	*O(log n)*		*O(log n)*	
`interval_sets`	*O(log n)*	*amortized O(log n)*		
`interval_maps`	*O(log n)*	*O(n)*	*O(log n)*	*O(n)*

Inplace operators

As presented in the overload tables for `operator -=` more type combinations are provided for subtraction than for addition.

```
// overload tables for             element containers:     interval containers:
T& operator -= (T&, const P&)      -= | e b s m            -= | e i b p S M
                                   ---+--------            ---+------------
                                   s  | s   s              S  | S S     S
                                   m  | m m m m            M  | M M M M M M
```

Subtraction provides the *reverse* operation of an addition for these overloads,

```
// Reverse addition                -= | e b s m            -= | e i b p S M
                                   ---+--------            ---+------------
                                   s  | s   s              S  | S S     S
                                   m  |   m   m            M  |     M M   M
```

and you can erase parts of `icl::maps` or `interval_maps` using *key values*, *intervals* or *element or interval sets* using these overloads:

```
// Erasure by key objects          -= | e b s m            -= | e i b p S M
                                   ---+--------            ---+------------
                                   s  | s   s              S  | S S     S
                                   m  | m   m              M  | M M     M
```

On Sets both function groups fall together as ***set difference***.

Complexity characteristics for inplace subtraction operations are given by the next tables where

```
n = iterative_size(y);
m = iterative_size(x); //if P is a container type
```

Table 25. Time Complexity for inplace Subtraction on element containers

`T& operator -= (T&, const P&)`	domain type	domain mapping type	std::set	icl::map
`std::set`	*O(log n)*		*O(m log n)*	
`icl::map`	*O(log n)*	*O(log n)*	*O(m log n)*	*O(m log n)*

Table 26. Time Complexity for inplace Subtraction on interval containers

`T& operator -= (T&, const P&)`	domain type	interval type	domain mapping type	interval mapping type	interval sets	interval maps
interval_sets	*O(log n)*	*amortized O(log n)*			*O(m log(n+m))*	
interval_maps	*O(log n)*	*amortized O(log n)*	*O(log n)*	*O(n)*	*O(m log(n+m))*	*O(m log(n+m))*

Infix operators

The admissible overloads for the infix *subtraction* `operator -` which is a non commutative operation is given by the next overload table.

```
// overload tables for          -  | e b s m       -  | e i b p S M
T operator - (T, const P&)      ---+--------       ---+------------
                                s  | s   s         S  | S S     S
                                m  | m m m m       M  | M M M M M M
```

Subtraction on Intervals

Subtraction	Types	Description

`T left_subtract(T right, const T& left_minuend)`	i	subtract `left_minuend` from the interval `right` on it's left side.
		`right_over = left_sub↵`
		`tract(right, left_minuend);`
		`...      d) : right`
		`... c)       : left_minuend`
		`     [c  d) : right_over`

`T right_subtract(T left, const T& right_minuend)`	i	subtract `right_minuend` from the interval `left` on it's right side.
		`left_over = right_sub↵`
		`tract(left, right_minuend);`
		`[a       ...  : left`
		`     [b ...  : right_minuend`
		`[a  b)       : left_over`

See also . . .

[Addition]
[Erasure]

Back to section . . .

Function Synopsis
Interface

Insertion

Synopsis

Insertion	interval sets	interval maps	element sets	element maps
`V T::insert(const P&)`	e i	b p	e	b
`V insert(T&, const P&)`	e i	b p	e	b
`V T::insert(J pos, const P&)`	i	p	e	b
`V insert(T&, J pos, const P&)`	i	p	e	b
`T& insert(T&, const P&)`	e i S	b p M	e s	b m
`T& T::set(const P&)`		b p		1
`T& set_at(T&, const P&)`		b p		1

Insertion

The effects of ***insertion*** implemented by `insert` and ***addition*** implemented by `add` and `operator +=` are identical for all Set-types of the **icl**.

For Map-types, `insert` provides the **stl** semantics of insertion in contrast to `add` and `operator +=`, that implement a generalized addition, that performs aggregations if key values collide or key intervals overlap. `insert` on Maps does not alter a maps content at the points, where the keys of the object to inserted overlap or collide with keys that are already in the map.

Setting values

Overwriting values using `operator[]` like in

```
my_map[key] = new_value;
```

is not provided for `interval_maps` because an `operator[]` is not implemented for them. As a substitute a function `T& T::set(const P&)` can be used to achieve the same effect:

```
my_map.set(make_pair(overwrite_this, new_value));
```

Insertion

```
// overload table for functions      T\P| e i b p
V T::insert(const P&)                ---+--------
V insert(T&, const P&)                s | s
                                      m |      m
                                      S |    S
                                      M |        M
```

Table 27. Time Complexity for member function insert on icl containers

`T& T::insert(const P&)`	domain type	interval type	domain mapping type	interval mapping type
`std::set`	*O(log n)*			
`icl::map`			*O(log n)*	
`interval_set` `separate_interval_set`	*O(log n)*	*amortized O(log n)*		
`split_interval_set`	*O(log n)*	*O(n)*		
`interval_map` `split_interval_map`			*O(log n)*	*O(n)*

```
// overload tables for function      element containers:     interval containers:
T& insert(T&, const P&)              T\P| e b s m            T\P| e i b p S M
                                     ---+--------            ---+------------
                                      s | s   s               S | S S     S
                                      m |   m   m             M |     M M   M
```

Table 28. Time Complexity for inplace insertion on element containers

`T& insert(T& y, const P& x)`	domain type	domain mapping type	interval sets	interval maps
`std::set`	*O(log n)*		*O(m)*	
`icl::map`		*O(log n)*		*O(m)*

Time complexity characteristics of inplace insertion for interval containers is given by this table.

Table 29. Time Complexity for inplace insertion on interval containers

`T& insert(T& y, const P& x)`		domain type	interval type	domain mapping type	interval mapping type	interval sets	interval maps
interval_sets	`interval_set` `separate_interval_set`	*O(log n)*	*amortized O(log n)*			*O(m log(n+m))*	
	`split_interval_set`	*O(log n)*	*O(n)*			*O(m log(n+m))*	
interval_maps				*O(log n)*	*O(n)*		*O(m log(n+m))*

Hinted insertion

Function `T& T::insert(T::iterator prior, const P& addend)` allows for an insertion in ***constant time***, if `addend` can be inserted right after iterator `prior` without collision. If this is not possible the complexity characteristics are as stated for the non hinted insertion above. Hinted insertion is available for these combinations of types:

```
// overload table for insertion with hint      T\P| e i b p
V T::insert(J pos, const P&)                   ---+--------
V insert(T&, J pos, const P&)                   s | s
                                                m |     m
                                                S |   S
                                                M |       M
```

Setting values

```
// overload table for member function          T\P| b p
T& T::set(const P&)                            ---+----
T& set_at(T&, const P&)                         m | m
                                                M |   M
```

Table 30. Time Complexity for member function `set`

`T& set(T&, const P&)`	domain_mapping_type	interval_mapping_type
icl::map	*O(log n)*	
interval_maps		*amortized O(log n)*

See also . . .

Erasure
Addition

Back to section . . .

Function Synopsis
Interface

Erasure

Synopsis

Erasure	interval sets	interval maps	element sets	element maps
`T& T::erase(const P&)`	**e i**	**e i b p**	**e**	**b p**
`T& erase(T&, const P&)`	**e i S**	**e i S b p M**	**e s**	**b m**
`void T::erase(iterator)`	1	1	1	1
`void T::erase(iterator,iterator)`	1	1	1	1

Erasure

The effects of ***erasure*** implemented by `erase` and ***subtraction*** implemented by `subtract` and `operator -=` are identical for all Set-types of the **icl**.

For Map-types, `erase` provides the **stl** semantics of erasure in contrast to `subtract` and `operator -=`, that implement a generalized subtraction, that performs inverse aggregations if key values collide or key intervals overlap.

Using iterators it is possible to erase objects or ranges of objects the iterator is pointing at from icl Sets and Maps.

Erasure of Objects

```
/* overload table for */       T\P| e i b p
T& T::erase(const P&)          ---+--------
T& erase(T&, const P&)           s | s
                                 m |     m
                                 S | S S
                                 M |     M M
```

The next table contains complexity characteristics for the `erase` function on elements and segments.

Table 31. Time Complexity for erasure of elements and segments on icl containers

`T& T::erase(const P&)` `T& erase(T&, const P&)`	domain type	interval type	domain mapping type	interval mapping type
`std::set`	*O(log n)*			
`icl::map`	*O(log n)*		*O(log n)*	
`interval_sets`	*O(log n)*	*amortized O(log n)*		
`interval_maps`	*O(log n)*	*O(n)*	*O(log n)*	*O(n)*

As presented in the overload tables for inplace function `erase` below, more type combinations are available for *erasure* than for *insertion*.

```
// overload tables for function    element containers:     interval containers:
T& erase(T&, const P&)             T\P| e b s m            T\P| e i b p S M
                                   ---+--------            ---+------------
                                    s | s   s               S | S S     S
                                    m | m m m m             M | M M M M M M
```

We can split up these overloads in two groups. The first group can be called *reverse insertion*.

```
/* (1) Reverse insertion */        T\P| e b s m            T\P| e i b p S M
                                   ---+--------            ---+------------
                                    s | s   s               S | S S     S
                                    m |   m   m             M |     M M   M
```

The second group can be viewed as an *erasure by key objects*

```
/* (2) Erasure by key objects */   T\P| e b s m            T\P| e i b p S M
                                   ---+--------            ---+------------
                                    s | s   s               S | S S     S
                                    m | m   m               M | M M     M
```

On Maps ***reverse insertion (1)*** is different from **stl's** erase semantics, because value pairs are deleted only, if key ***and*** data values are found. Only ***erasure by key objects (2)*** works like the erase function on **stl's** std::maps, that passes a ***key value*** as argument.

On Sets both function groups fall together as ***set difference***.

Complexity characteristics for inplace erasure operations are given by the next tables where

```
n = iterative_size(y);
m = iterative_size(x); //if P is a container type
```

Table 32. Time Complexity for inplace erasure on element containers

`T& erase(T& y, const P& x)`	domain type	domain mapping type	std::set	icl::map
`std::set`	*O(log n)*		*O(m log n)*	
`icl::map`	*O(log n)*	*O(log n)*	*O(m log n)*	*O(m log n)*

Table 33. Time Complexity for inplace erasure on interval containers

`T& erase(T& y, const P& x)`	domain type	interval type	domain mapping type	interval mapping type	interval sets	interval maps
interval_sets	*O(log n)*	*amortized O(log n)*			*O(m log(n+m))*	
interval_maps	*O(log n)*	*amortized O(log n)*	*O(log n)*	*O(n)*	*O(m log(n+m))*	*O(m log(n+m))*

Erasure by Iterators

The next table shows the **icl** containers that erasure with iterators is available for. Erase on iterators erases always one `value` of `value_type` for an iterator pointing to it. So we erase

- elements from `std::sets`
- element-value pairs from `icl::maps`
- intervals from `interval_sets` and
- interval-value-pairs from `interval_maps`

Erasure by iterators	interval sets	interval maps	element sets	element maps
`void T::erase(iterator pos)`	*amortized O(1)*	*amortized O(1)*	*amortized O(1)*	*amortized O(1)*
`void T::erase(iterator first, iterator past)`	*O(k)*	*O(k)*	*O(k)*	*O(k)*

Erasing by a single iterator need only ***amortized constant time***. Erasing via a range of iterators `[first, past)` is of ***linear time*** in the number `k` of iterators in range `[first, past)`.

See also . . .

Insertion
Subtraction

Back to section . . .

Function Synopsis
Interface

Intersection

Synopsis

Intersection	interval type	interval sets	interval maps	element sets	element maps
`void add_intersection(T&, const T&, const P&)`		e i S	e i S b p M		
`T& operator &=(T&, const P&)`		e i S	e i S b p M	e s	b m
`T operator & (T, const P&)` `T operator & (const P&, T)`	i	e i S	e i S b p M	e s	b m
`bool intersects(const T&, const P&)` `bool disjoint(const T&, const P&)`	i	e i S	e i S b p M	e s	b m

Functions and operators that are related to ***intersection*** on **icl** objects are given in the table above.

	Description of Intersection
`Sets`	Intersection on Sets implements ***set intersection***
`Maps`	Intersection on Maps implements a ***map intersection*** function similar to *set intersection*. If, on intersection, an element value pair `(k,v)` it's key `k` is in the map already, the intersection function is propagated to the associated value, if it exists for the Map's codomain_type. If the codomain_type has no intersection operation, associated values are combined using addition. For partial map types this results in an addition on the intersection of the domains of the intersected sets. For total maps intersection and addition are identical in this case. See also *intersection on Maps of numbers*. A Map can be intersected with key types: an element (an interval for interval_maps) and and a Set. This results in the selection of a submap, and can be defined as a generalized selection function on Maps.

Functions

The overloaded function

`void add_intersection(T& result, const T& y, const P& x)`

allows to accumulate the intersection of `y` and `x` in the first argument `result`. `Result` might already contain data. In this case the intersection of `y` and `x` is `added` the the contents of `result`.

```
T s1 = f, s2 = f, y = g; P x = h; // The effect of
add_intersection(s1, y, x);       // add_intersection
s2 += (y & x);                    // and & followed by +=
assert(s1==s2);                   // is identical
```

This might be convenient, if intersection is used like a generalized selection function. Using element or segment types for `P`, we can select small parts of a container `y` and accumulate them in `section`.

The admissible combinations of types for function `void add_intersection(T&, const T&, const P&)` can be summarized in the ***overload table*** below. Compared to other overload tables, placements of function arguments are different: Row headers denote type `T` of `*this` object. Columns headers denote type `P` of the second function argument. The table cells contain the arguments `T` of the intersections `result`, which is the functions first argument.

```
/* overload table for */                                T\P| e i b p
void T::add_intersection(T& result, const P&)const      ---+--------
                                                         s | s
                                                         m | m   m
                                                         S | S S
                                                         M | M M M M
```

The next table contains complexity characteristics for function `add_intersection`.

Table 34. Time Complexity for function add_intersection on icl containers

`void add_intersection(T&, const T&, const P&)const`	domain type	interval type	domain mapping type	interval mapping type
`std::set`	*O(log n)*			
`icl::map`	*O(log n)*		*O(log n)*	
`interval_sets`	*O(log n)*	*O(n)*		
`interval_maps`	*O(log n)*	*O(n)*	*O(n)*	*O(n)*

Inplace operators

The overload tables below are giving admissible type combinations for the intersection `operator &=`. As for the overload patterns of *subtraction* intersections are possible within Sets and Maps but also for Maps combined with *key objects* which are *key elements*, *intervals* and *Sets of keys*.

```
// overload tables for             element containers:     interval containers:
T& operator &= (T&, const P&)      &= | e b s m            &= | e i b p S M
                                   ---+--------            ---+------------
                                   s  | s   s              S  | S S     S
                                   m  | m m m m            M  | M M M M M M
```

While intersection on maps can be viewed as a ***generalisation of set intersection***. The combination on Maps and Sets can be interpreted as a ***generalized selection function***, because it allows to select parts of a maps using *key* or *selection objects*. So we have a ***generalized intersection*** for these overloads,

```
/* (Generalized) intersection */   &= | e b s m            &= | e i b p S M
                                   ---+--------            ---+------------
                                   s  | s   s              S  | S S     S
                                   m  |   m   m            M  |     M M   M
```

and a ***selection by key objects*** here:

```
/* Selection by key objects */     &= | e b s m            &= | e i b p S M
                                   ---+--------            ---+------------
                                   s  | s   s              S  | S S     S
                                   m  | m   m              M  | M M     M
```

The differences for the different functionalities of `operator &=` are on the Map-row of the tables. Both functionalities fall together for Sets in the function ***set intersection***.

Complexity characteristics for inplace intersection operations are given by the next tables where

```
n = iterative_size(y);
m = iterative_size(x); //if P is a container type
```

Table 35. Time Complexity for inplace intersection on element containers

`T& operator &= (T& y, const P& x)`	**domain type**	**domain mapping type**	**std::set**	**icl::map**
`std::set`	*O(log n)*		*O(m log n)*	
`icl::map`	*O(log n)*	*O(log n)*	*O(m log n)*	*O(m log n)*

Table 36. Time Complexity for inplace intersection on interval containers

`T& operator &= (T& y, const P& x)`	**domain type**	**interval type**	**domain mapping type**	**interval mapping type**	**interval sets**	**interval maps**
interval_sets	*O(log n)*	*O(n)*			*O(m log(n+m))*	
interval_maps	*O(log n)*	*O(n)*	*O(log n)*	*O(n)*	*O(m log(n+m))*	*O(m log(n+m))*

Infix operators

For the **icl's** infix intersection the following overloads are available:

```
// overload tables for              element containers:     interval containers:
T operator & (T, const P&)          &  | e b s m            &  | e  i  b  p  S1 S2 S3 M1 M3
T operator & (const P&, T)          ---+--------            ---+--------------------------
                                    e  |     s m            e  |             S1 S2 S3 M1 M3
                                    b  |       m            i  |    i        S1 S2 S3 M1 M3
                                    s  | s   s m            b  |                      M1 M3
                                    m  | m m m m            p  |                      M1 M3
                                                            S1 | S1 S1       S1 S2 S3 M1 M3
                                                            S2 | S2 S2       S2 S2 S3 M1 M3
                                                            S3 | S3 S3       S3 S3 S3 M1 M3
                                                            M1 | M1 M1 M1 M1 M1 M1 M1 M1 M3
                                                            M3 | M3 M3 M3 M3 M3 M3 M3 M3 M3
```

To resolve ambiguities among interval containers the ***finer*** container type is chosen as result type.

Again, we can split up the overload tables of `operator &` in a part describing the **generalized intersection* on interval containers and a second part defining the **selection by key object* functionality.

```
/* (Generalized) intersection */    &  | e b s m            &  | e  i  b  p  S1 S2 S3 M1 M3
                                    ---+--------            ---+--------------------------
                                    e  |     s              e  |             S1 S2 S3
                                    b  |       m            i  |    i        S1 S2 S3
                                    s  | s   s              b  |                      M1 M3
                                    m  |   m   m            p  |                      M1 M3
                                                            S1 | S1 S1       S1 S2 S3
                                                            S2 | S2 S2       S2 S2 S3
                                                            S3 | S3 S3       S3 S3 S3
                                                            M1 |       M1 M1          M1 M3
                                                            M3 |       M3 M3          M3 M3
```

```
/* Selection by key objects */      &   | e b s m           &   | e  i  b  p  S1 S2 S3 M1 M3
                                    ---+--------           ---+--------------------------
                                    e   |     s m           e   |             S1 S2 S3 M1 M3
                                    b   |                   i   |    i        S1 S2 S3 M1 M3
                                    s   | s   s m           b   |
                                    m   | m   m             p   |
                                                            S1  | S1 S1       S1 S2 S3 M1 M3
                                                            S2  | S2 S2       S2 S2 S3 M1 M3
                                                            S3  | S3 S3       S3 S3 S3 M1 M3
                                                            M1  | M1 M1       M1 M1 M1
                                                            M3  | M3 M3       M3 M3 M3
```

Intersection tester

Tester	Desctription
`bool intersects(const T& left, const P& right)`	Tests, if `left` and `right` intersect.
`bool disjoint(const T& left, const P& right)`	Tests, if `left` and `right` are disjoint.
	`intersects(x,y) == !disjoint(x,y)`

```
bool intersects(const T&, const P&)      T\P| e b s m      T\P| e i b p S M
bool   disjoint(const T&, const P&)      ---+--------      ---+------------
                                          s | 1   1         S | 1 1     1
                                          m | 1 1 1 1       M | 1 1 1 1 1 1
```

See also . . .

Symmetric difference
Subtraction
Addition

Back to section . . .

Function Synopsis
Interface

Symmetric Difference

Synopsis

Symmetric difference	interval sets	interval maps	element sets	element maps
`T& T::flip(const P&)`	**e i**	**b p**		**b**
`T& flip(T&, const P&)`	**e i**	**b p**	**e**	**b**
`T& operator ^=(T&, const P&)`	**e i S**	**b p M**	**e s**	**b m**
`T operator ^ (T, const P&)` `T operator ^ (const P&, T)`	**e i S**	**b p M**	**e s**	**b m**

Functions and operators that implement ***symmetric difference*** on **icl** objects are given in the table above.

	Description of symmetric difference
`Sets`	`operator ^` implements ***set symmetric difference***
`Maps`	`operator ^` implements a ***map symmetric difference*** function similar to *set symmetric difference*. All pairs that are common to both arguments are removed. All others unified.

Functions

Symmetric difference is implemented on interval containers by the function `T& flip(T&, const P& operand)`.

```
flip(y,x)
```

deletes every element of `y`, if it is contained in `x`. Elements of `x` not contained in `y` are added. For icl containers flip is also availabel as memeber function `T& T::flip(const P& operand)`.

The admissible combinations of types for member function `T& T::flip(const P&)` can be summarized in the ***overload table*** below:

```
/* overload table for */           T\P| e i b p
T& T::flip(const P&)               ---+--------
T& flip(T&, const P&)               s | s
                                    m |     m
                                    S | S S
                                    M |     M M
```

The next table contains complexity characteristics for functions `flip`.

Table 37. Time Complexity for member functions flip on icl containers

`T& T::flip(const P&)` `T& flip(T&, const P&)`	domain type	interval type	domain mapping type	interval mapping type
`std::set`	*O(log n)*			
`icl::map`			*O(log n)*	
`interval_set` `separate_interval_set`	*O(log n)*	*O(n)*		
`split_interval_set`	*O(log n)*	*O(n)*		
`interval_map` `split_interval_map`			*O(log n)*	*O(n)*

Inplace operators

The overload tables below are giving admissible type combinations for `operator ^=` that implements ***symmetric difference***.

```
// overload tables for             element containers:     interval containers:
T& operator ^= (T&, const P&)      ^= | e b s m            ^= | e i b p S M
                                   ---+--------            ---+------------
                                   s  | s   s              S  | S S     S
                                   m  |   m   m            M  |     M M   M
```

Complexity characteristics for inplace operators that implement ***symmetric difference*** are given by the next tables where

```
n = iterative_size(y);
m = iterative_size(x); //if P is a container
```

Table 38. Time Complexity for inplace symmetric difference on element containers

`T& operator &= (T& y, const P& x)`	domain type	domain mapping type	std::set	icl::map
`std::set`	*O(log n)*		*O(m log n)*	
`icl::map`	*O(log n)*	*O(log n)*	*O(m log n)*	*O(m log n)*

Table 39. Time Complexity for inplace symmetric difference on interval containers

`T& operator &= (T&, const P&)`	domain type	interval type	domain mapping type	interval mapping type	interval sets	interval maps
interval_sets	*O(log n)*	*O(n)*			*O(m log(n+m))*	
interval_maps	*O(log n)*	*O(n)*	*O(log n)*	*O(n)*	*O(m log(n+m))*	*O(m log(n+m))*

Infix operators

For the infix version of symmetric difference the following overloads are available:

```
// overload tables for             element containers:     interval containers:
T operator ^ (T, const P&)         ^  | e b s m            ^  | e  i  b  p  S1 S2 S3 M1 M3
T operator ^ (const P&, T)         ---+--------            ---+---------------------------
                                   e  |     s              e  |             S1 S2 S3
                                   b  |       m            i  |             S1 S2 S3
                                   s  | s   s              b  |                      M1 M3
                                   m  |   m   m            p  |                      M1 M3
                                                           S1 | S1 S1       S1 S2 S3
                                                           S2 | S2 S2       S2 S2 S3
                                                           S3 | S3 S3       S3 S3 S3
                                                           M1 |       M1 M1          M1 M3
                                                           M3 |       M3 M3          M3 M3
```

To resolve ambiguities among interval containers the ***finer*** container type is chosen as result type.

See also . . .

Intersection
Subtraction
Addition

Back to section . . .

Function Synopsis
Interface

Iterator related

Synopsis Complexities	**interval sets**	**interval maps**	**element sets**	**element maps**
`J T::begin()`	*O(1)*	*O(1)*	*O(1)*	*O(1)*
`J T::end()`	*O(1)*	*O(1)*	*O(1)*	*O(1)*
`J T::rbegin()`	*O(1)*	*O(1)*	*O(1)*	*O(1)*
`J T::rend()`	*O(1)*	*O(1)*	*O(1)*	*O(1)*
`J T::lower_bound(const key_type&)`	*O(log n)*	*O(log n)*	*O(log n)*	*O(log n)*
`J T::upper_bound(const key_type&)`	*O(log n)*	*O(log n)*	*O(log n)*	*O(log n)*
`pair<J,J> T::equal_range(const key_type&)`	*O(log n)*	*O(log n)*	*O(log n)*	*O(log n)*

Iterator related	
`iterator T::begin()` `const_iterator T::begin()const`	Returns an iterator to the first value of the container.
`iterator T::end()` `const_iterator T::end()const`	Returns an iterator to a position `end()` after the last value of the container.
`reverse_iterator T::rbegin()` `const_reverse_iterator T::rbegin()const`	Returns a reverse iterator to the last value of the container.
`reverse_iterator T::rend()` `const_reverse_iterator T::rend()const`	Returns a reverse iterator to a position `rend()` before the first value of the container.
`iterator T::lower_bound(const key_type& k)` `const_iterator T::lower_bound(const key_type& key)const`	Returns an iterator that points to the first element `first`, that does not compare less than `key_type key`. `first` can be equal or greater than `key`, or it may overlap `key` for interval containers.
`iterator T::upper_bound(const key_type&)` `const_iterator T::upper_bound(const key_type&)const`	Returns an iterator that points to the first element `past`, that compares greater than `key_type key`.
`pair<iterator,iterator> T::equal_range(const key_type& key)` `pair<const_iterator,const_iterator> T::equal_range(const key_type& key)const`	Returns a range `[first, past)` of iterators to all elements of the container that compare neither less than nor greater than `key_type key`. For element containers `std::set` and `icl::map`, `equal_range` contains at most one iterator pointing the element equal to `key`, if it exists. For interval containers `equal_range` contains iterators to all intervals that overlap interval `key`.

See also . . .

Element iteration

Back to section . . .

Function Synopsis
Interface

Element iteration

This section refers to ***element iteration*** over ***interval containers***. Element iterators are available as associated types on interval sets and interval maps.

Variant	Associated element iterator type for interval container `T`
forward	`T::element_iterator`
const forward	`T::element_const_iterator`
reverse	`T::element_reverse_iterator`
const reverse	`T::element_const_reverse_iterator`

There are also associated iterators types `T::iterator`, `T::const_iterator`, `T::reverse_iterator` and `T::reverse_const_iterator` on interval containers. These are ***segment iterators***. Segment iterators are "first citizen iterators". Iteration over segments is fast, compared to an iteration over elements, particularly if intervals are large. But if we want to view our interval containers as containers of elements that are usable with std::algoritms, we need to iterate over elements.

Iteration over elements . . .

- is possible only for integral or discrete `domain_types`
- can be very ***slow*** if the intervals are very large.
- and is therefore ***depreciated***

On the other hand, sometimes iteration over interval containers on the element level might be desired, if you have some interface that works for `std::SortedAssociativeContainers` of elements and you need to quickly use it with an interval container. Accepting the poorer performance might be less bothersome at times than adjusting your whole interface for segment iteration.

Caution

So we advice you to choose element iteration over interval containers ***judiciously***. Do not use element iteration ***by default or habitual***. Always try to achieve results using member functions, global functions or operators (preferably inplace versions) or iteration over segments first.

Synopsis Complexities	**interval sets**	**interval maps**
`J elements_begin(T&)`	*O(1)*	*O(1)*
`J elements_end(T&)`	*O(1)*	*O(1)*
`J elements_rbegin(T&)`	*O(1)*	*O(1)*
`J elements_rend(T&)`	*O(1)*	*O(1)*

Element iteration	Description
`element_iterator elements_begin(T&)` `element_const_iterator elements_begin(const T&)`	Returns an element iterator to the first element of the container.
`element_iterator elements_end(T&)` `element_const_iterator elements_end(const T&)`	Returns an element iterator to a position `elements_end(c)` after the last element of the container.
`element_reverse_iterator elements_rbegin(T&)` `element_const_reverse_iterator elements_rbegin(const T&)`	Returns a reverse element iterator to the last element of the container.
`element_reverse_iterator elements_rend(T&)` `element_const_reverse_iterator elements_rend(const T&)`	Returns a reverse element iterator to a position `elements_rend(c)` before the first element of the container.

Example

```
interval_set<int> inter_set;
inter_set.add(interval<int>::right_open(0,3))
         .add(interval<int>::right_open(7,9));

for(interval_set<int>::element_const_iterator creeper = elements_begin(inter_set);
    creeper != elements_end(inter_set); ++creeper)
    cout << *creeper << " ";
cout << endl;
//Program output: 0 1 2 7 8

for(interval_set<int>::element_reverse_iterator repeerc = elements_rbegin(inter_set);
    repeerc != elements_rend(inter_set); ++repeerc)
    cout << *repeerc << " ";
cout << endl;
//Program output: 8 7 2 1 0
```

See also . . .

Segment iteration

Back to section . . .

Function Synopsis
Interface

Streaming, conversion

Streaming, conversion	**intervals**	**interval sets**	**interval maps**	**element sets**	**element maps**
`std::basic_ostream operator << (basic_ostream&, const T&)`	1	1	1	1	1

Streaming, conversion	**Description**
`std::basic_ostream operator << (basic_ostream&, const T&)`	Serializes the argument of type T to an output stream

Back to section . . .

Function Synopsis
Interface

Interval Construction

T	**discrete _interval**	**continuous _interval**	**right_open _interval**	**left_open _interval**	**closed _interval**	**open _interval**
Interval bounds	dynamic	dynamic	static	static	static	static
Form			asymmetric	asymmetric	symmetric	symmetric
Construct						
`T singleton(const P&)`	**d**	**c**	**d**	**d**	**d**	**d**
`T construct(const P&, const P&)`	**d**	**c**	**d c**	**d c**	**d**	**d**
`T construct(const P&, const P&, interval_bounds )`	**d**	**c**				
`T hull(const P&, const P&)`	**d**	**c**	**d c**	**d c**	**d**	**d**
`T span(const P&, const P&)`	**d**	**c**	**d c**	**d c**	**d**	**d**
`static T right_open(const P&, const P&)`	**d**	**c**				
`static T left_open(const P&, const P&)`	**d**	**c**				
`static T closed(const P&, const P&)`	**d**	**c**				
`static T open(const P&, const P&)`	**d**	**c**				

The table above shows the availability of functions, that allow the construction of intervals. All interval constructin functins are of ***constant time and space complexity***.

<table>
<tr><th>Construct</th><th>Description</th></tr>
<tr><td><code>T singleton(const P& value)</code></td><td>Constructs an interval that contains exactly one element <code>value</code>. For all interval types of the icl sigletons can be constructed for discrete domain types. For continuous domain types, only <code>continuous_interval</code> is capable to construct a singleton.</td></tr>
<tr><td><code>T construct(const P& lower, const P& upper)</code></td><td>Contructs an interval with lower bound <code>lower</code> and upper bound <code>upper</code></td></tr>
<tr><td><pre>T construct(const P& lower, const P& upper,
            interval_bounds bounds
            = interval_bounds::right_open())</pre></td><td>For dynamically bounded intervals this function constructs an interval with interval bounds specified by the third parameter.</td></tr>
<tr><td><code>T hull(const P& x1, const P& x2)</code></td><td><code>hull(x1,x2)</code> constructs the smallest interval that contains both <code>x1</code> and <code>x2</code>. <code>x2</code> may be smaller than <code>x1</code>.</td></tr>
<tr><td><code>T span(const P& x1, const P& x2)</code></td><td><code>span(x1,x2)</code> constructs the interval <code>construct(min(x1,x2), max(x1,x2))</code>. Note the differences between <code>span</code>, <code>hull</code> and <code>construct</code>:
<pre>span<right_open_interval<int> >(2,1) == [1,2) // does NOT contain 2
hull<right_open_interval<int> >(2,1) == [1,3) // contains 2
construct<right_open_interval<int> >(2,1) == [)   // is empty</pre></td></tr>
<tr><td><pre>static T right_open(const P&, const P&)
static T left_open(const P&, const P&)
static T closed(const P&, const P&)
static T open(const P&, const P&)</pre></td><td>For dynamically bounded intervals there are for static functions to construct intervals with the four interval bound types:
<pre>discrete_interval<int>   itv1 = discrete_interval<int>::closed(0,42);
continuous_interval<double> itv2 = continuous_interval<double>::right_open(0.0, 1.0);</pre></td></tr>
<tr><td>Using the interval default</td><td></td></tr>
<tr><td><code>interval<P>::type</code></td><td>There is a library default, for all interval containers of the icl. The intension of the library default is to minimize the need for parameter specification, when working with icl class templates. We can get the library default interval type as <code>interval<P>::type</code>. The library default uses dynamically bounded intervals. You can switch to statically bounded intervals by <code>#define BOOST_ICL_USE_STATIC_BOUNDED_INTERVALS</code> prior to icl includes.</td></tr>
</table>

Construct	Description

`static T right_open(const P&, const P&)`
`static T left_open(const P&, const P&)`
`static T closed(const P&, const P&)`
`static T open(const P&, const P&)`
	For template struct `interval` that always uses the library default the static functions for the four interval bound types are also available.
	`interval<int>::type    itv1 = interval<int>::closed(0,42);`
	`interval<double>::type itv2 = interval<double>::right_open(0.0, 1.0);`
	This works with the statically bounded intervals as well, with the restriction that for continuous domain types the matching function has to be used:
	`#define BOOST_ICL_USE_STATIC_BOUNDED_INTERVALS`
	`. . .`
	`// library default is the statically bounded right_open_interval`
	`interval<int>::type    itv1 = interval<int>::closed(0,42); //==[0,43) //ok, bounds are shifted`
	`interval<double>::type itv2 = interval<double>::right_open(0.0, 1.0); //ok. right_open matches`
	`interval<double>::type itv3 = interval<double>::closed(0.0, 1.0);     //will NOT compile`
	See also examples Dynamic intervals and Static intervals

See also . . .

Example: Dynamically bounded intervals and the library default
Example: Statically bounded intervals, changing the library default

Back to section . . .

Additional interval functions
Function Synopsis
Interface

Additional Interval Orderings

In addition to the standard orderings `operator <` and rleated `> <= >=` that you will find in the ***librarie's function synopsis***, intervals implement some additional orderings that can be useful.

T	discrete _interval	continuous _interval	right_open _interval	left_open _interval	closed _interval	open _interval
Interval bounds	dynamic	dynamic	static	static	static	static
Form			asymmetric	asymmetric	symmetric	symmetric
Orderings						
`bool exclusive_less(const T&, const T&)`	1	1	1	1	1	1

`bool lower_less(const T&, const T&)`
`bool lower_equal(const T&, const T&)`
`bool lower_less_equal(const T&, const T&)`
	1	1	1	1	1	1

`bool upper_less(const T&, const T&)`
`bool upper_equal(const T&, const T&)`
`bool upper_less_equal(const T&, const T&)`
	1	1	1	1	1	1

A central role for the **icl** plays the `exclusive_less` ordering, which is used in all interval containers. The other orderings can be useful to simplify comparison of intervals specifically for dynamically bounded ones.

Orderings	**Description**
`bool exclusive_less(const T&, const T&)`	`exclusive_less(x1, x2)` is true if every element of interval `x1` is less than every element of interval `x2` w.r.t. the the intervals `Compare` ordering

`bool lower_less(const T&, const T&)`
`bool lower_equal(const T&, const T&)`
`bool lower_less_equal(const T&, const T&)`
	Compares the beginnings of intervals.
	`lower_less(x,y)  == true; // x begins before y`
	`lower_equal(x,y) == true; // x and y begin at the same element`
	`lower_less_equal(x,y) == lower_less(x,y) || lower_equal(x,y);`

`bool upper_less(const T&, const T&)`
`bool upper_equal(const T&, const T&)`
`bool upper_less_equal(const T&, const T&)`
	Compares the endings of intervals.
	`upper_less(x,y)  == true; // x ends before y`
	`upper_equal(x,y) == true; // x and y end at the same element`
	`upper_less_equal(x,y) == upper_less(x,y) || upper_equal(x,y);`

See also . . .

Equivalences and Orderings

Back to section . . .

Additional interval functions
Function Synopsis
Interface

Miscellaneous Interval Functions

T	**discrete _interval**	**continuous _interval**	**right_open _interval**	**left_open _interval**	**closed _interval**	**open _interval**
Interval bounds	dynamic	dynamic	static	static	static	static
Form			asymmetric	asymmetric	symmetric	symmetric
Miscellaneous						
`bool touches(const T&, const T&)`	1	1	1	1	1	1
`T inner_complement(const T&, const T&)`	1	1	1	1	1	1
`difference_type distance(const T&, const T&)`	1	1	1	1	1	1

Miscellaneous Interval Functions	**Description**
`bool touches(const T&, const T&)`	`touches(x,y)` Between the disjoint intervals `x` and `y` are no elements.
`T inner_complement(const T&, const T&)`	`z = inner_complement(x,y)` `z` is the interval between `x` and `y`
`difference_type distance(const T&, const T&)`	`d = distance(x,y)` If the domain type of the interval has a difference_type, `d` is the distance between `x` and `y`.

Back to section . . .

Additional interval functions
Function Synopsis
Interface

Acknowledgments

I would like to thank CEO Hermann Steppe and Chief Developer Peter Wuttke of Cortex Software GmbH for their friendly support of my work on the Icl and their permission to release the library as open source. For her genuine interest in my work and many creative and uplifting talks I want to thank my colleague Axinja Ott who contributed a lot to the success of the project.

Many thanks to Paul A. Bristow, Vicente Botet, Luke Simonson, Alp Mestan, David Abrahams, Steven Watanabe, Neal Becker, Phil Endecott, Robert Stewart, Jeff Flinn, Zachary Turner and other developers form the Boost community who supported the development of the Interval Template Library by numerous hints and feedbacks on the boost mailing list. Also helpful have been conversations, hints and contributions at the BoostCon 2009 by Jeff Garland, David Jenkins, Tobias Loew, Barend Gehrels, Luke Simonson and Hartmut Kaiser. Special thanks for reading and improving this documentation to Neal Becker, Ilya Bobir and Brian Wood. Jeff Flinn provided valuable feedback and a codepatch to fix portability problems with CodeWarrior 9.4. Many thanks for that.

I am grateful to Hartmut Kaiser for managing the formal review of this library and to all the reviewers and participants in the related discussions, including Jeff Flinn, Luke Simonson, Phil Endecott, Eric M. Jonas, Peter Wuttke, Robert Stewart, Barend Gehrels, Vicente Botet, Thomas Klimpel, Paul A. Bristow, Jerry Jeremiah, John Reid, Steven Watanabe, Brian Wood, Markus Werle and Michael Caisse.

For sending in bug reports, code patches and suggestions for improvements after the review thanks to Denis and Marvin Sielenkemper.

Interval Container Library Reference

Header <boost/icl/closed_interval.hpp>

```
namespace boost {
  namespace icl {
    template<typename DomainT, 
             ICL_COMPARE Compare = ICL_COMPARE_INSTANCE(ICL_COMPARE_DEFAULT, DomainT)> 
      class closed_interval;

    template<typename DomainT, ICL_COMPARE Compare> 
      struct interval_traits<icl::closed_interval< DomainT, Compare >>;
    template<typename DomainT, ICL_COMPARE Compare> 
      struct interval_bound_type<closed_interval< DomainT, Compare >>;
    template<typename DomainT, ICL_COMPARE Compare> 
      struct type_to_string<icl::closed_interval< DomainT, Compare >>;
    template<typename DomainT> 
      struct value_size<icl::closed_interval< DomainT >>;
  }
}
```

Class template closed_interval

boost::icl::closed_interval

Synopsis

```
// In header: <boost/icl/closed_interval.hpp>

template<typename DomainT, 
         ICL_COMPARE Compare = ICL_COMPARE_INSTANCE(ICL_COMPARE_DEFAULT, DomainT)> 
class closed_interval {
public:
  // types
  typedef closed_interval< DomainT, Compare > type;       
  typedef DomainT                             domain_type;

  // construct/copy/destruct
  closed_interval();
  explicit closed_interval(const DomainT &);
  closed_interval(const DomainT &, const DomainT &);

  // public member functions
  typedef ICL_COMPARE_DOMAIN(Compare, DomainT);
  DomainT lower() const;
  DomainT upper() const;
  DomainT first() const;
  DomainT last() const;
};
```

Description

`closed_interval` public construct/copy/destruct

1.
```
closed_interval();
```

Default constructor; yields an empty interval `[0,0)`.

2.
```
explicit closed_interval(const DomainT & val);
```

Constructor for a closed singleton interval `[val,val]`

3.
```
closed_interval(const DomainT & low, const DomainT & up);
```

Interval from `low` to `up` with bounds `bounds`

`closed_interval` public member functions

1.
```
typedef ICL_COMPARE_DOMAIN(Compare, DomainT);
```

2.
```
DomainT lower() const;
```

3.
```
DomainT upper() const;
```

4.
```
DomainT first() const;
```

5.
```
DomainT last() const;
```

Struct template interval_traits<icl::closed_interval< DomainT, Compare >>

boost::icl::interval_traits<icl::closed_interval< DomainT, Compare >>

Synopsis

```
// In header: <boost/icl/closed_interval.hpp>

template<typename DomainT, ICL_COMPARE Compare> 
struct interval_traits<icl::closed_interval< DomainT, Compare >> {
  // types
  typedef DomainT                                    domain_type;  
  typedef icl::closed_interval< DomainT, Compare > interval_type;

  // public member functions
  typedef ICL_COMPARE_DOMAIN(Compare, DomainT);

  // public static functions
  static interval_type construct(const domain_type &, const domain_type &);
  static domain_type lower(const interval_type &);
  static domain_type upper(const interval_type &);
};
```

Description

`interval_traits` public member functions

1.
```
typedef ICL_COMPARE_DOMAIN(Compare, DomainT);
```

`interval_traits` public static functions

1. ```
   static interval_type construct(const domain_type & lo, const domain_type & up);
   ```

2. ```
   static domain_type lower(const interval_type & inter_val);
   ```

3. ```
   static domain_type upper(const interval_type & inter_val);
   ```

Struct template interval_bound_type<closed_interval< DomainT, Compare >>

boost::icl::interval_bound_type<closed_interval< DomainT, Compare >>

Synopsis

```
// In header: <boost/icl/closed_interval.hpp>

template<typename DomainT, ICL_COMPARE Compare>
struct interval_bound_type<closed_interval< DomainT, Compare >> {
  // types
  typedef interval_bound_type type;

  // public member functions
   BOOST_STATIC_CONSTANT(bound_type, value = interval_bounds::static_closed);
};
```

Description

`interval_bound_type` public member functions

1. ```
    BOOST_STATIC_CONSTANT(bound_type, value = interval_bounds::static_closed);
   ```

Struct template type_to_string<icl::closed_interval< DomainT, Compare >>

boost::icl::type_to_string<icl::closed_interval< DomainT, Compare >>

Synopsis

```
// In header: <boost/icl/closed_interval.hpp>

template<typename DomainT, ICL_COMPARE Compare>
struct type_to_string<icl::closed_interval< DomainT, Compare >> {

  // public static functions
  static std::string apply();
};
```

Description

`type_to_string` public static functions

1.
```
static std::string apply();
```

Struct template value_size<icl::closed_interval< DomainT >>

boost::icl::value_size<icl::closed_interval< DomainT >>

Synopsis

```
// In header: <boost/icl/closed_interval.hpp>

template<typename DomainT>
struct value_size<icl::closed_interval< DomainT >> {

  // public static functions
  static std::size_t apply(const icl::closed_interval< DomainT > &);
};
```

Description

`value_size` public static functions

1.
```
static std::size_t apply(const icl::closed_interval< DomainT > &);
```

Header <boost/icl/continuous_interval.hpp>

```
namespace boost {
  namespace icl {
    template<typename DomainT,
             ICL_COMPARE Compare = ICL_COMPARE_INSTANCE(ICL_COMPARE_DEFAULT, DomainT)>
      class continuous_interval;

    template<typename DomainT, ICL_COMPARE Compare>
      struct interval_traits<icl::continuous_interval< DomainT, Compare >>;
    template<typename DomainT, ICL_COMPARE Compare>
      struct dynamic_interval_traits<boost::icl::continuous_interval< DomainT, Compare >>;
    template<typename DomainT, ICL_COMPARE Compare>
      struct interval_bound_type<continuous_interval< DomainT, Compare >>;
    template<typename DomainT, ICL_COMPARE Compare>
      struct is_continuous_interval<continuous_interval< DomainT, Compare >>;
    template<typename DomainT, ICL_COMPARE Compare>
      struct type_to_string<icl::continuous_interval< DomainT, Compare >>;
    template<typename DomainT>
      struct value_size<icl::continuous_interval< DomainT >>;
  }
}
```

Class template continuous_interval

boost::icl::continuous_interval

Synopsis

```
// In header: <boost/icl/continuous_interval.hpp>

template<typename DomainT, 
         ICL_COMPARE Compare = ICL_COMPARE_INSTANCE(ICL_COMPARE_DEFAULT, DomainT)> 
class continuous_interval {
public:
  // types
  typedef continuous_interval< DomainT, Compare > type;              
  typedef DomainT                                 domain_type;       
  typedef bounded_value< DomainT >::type          bounded_domain_type;

  // construct/copy/destruct
  continuous_interval();
  explicit continuous_interval(const DomainT &);
  continuous_interval(const DomainT &, const DomainT &, 
                      interval_bounds = interval_bounds::right_open(), 
                      continuous_interval * = 0);

  // public member functions
  typedef ICL_COMPARE_DOMAIN(Compare, DomainT);
  domain_type lower() const;
  domain_type upper() const;
  interval_bounds bounds() const;

  // public static functions
  static continuous_interval open(const DomainT &, const DomainT &);
  static continuous_interval right_open(const DomainT &, const DomainT &);
  static continuous_interval left_open(const DomainT &, const DomainT &);
  static continuous_interval closed(const DomainT &, const DomainT &);
};
```

Description

`continuous_interval` public construct/copy/destruct

1.
```
continuous_interval();
```

   Default constructor; yields an empty interval `[0,0)`.

2.
```
explicit continuous_interval(const DomainT & val);
```

   Constructor for a closed singleton interval `[val,val]`

3.
```
continuous_interval(const DomainT & low, const DomainT & up, 
                    interval_bounds bounds = interval_bounds::right_open(), 
                    continuous_interval * = 0);
```

   Interval from `low` to `up` with bounds `bounds`

`continuous_interval` public member functions

1.
```
typedef ICL_COMPARE_DOMAIN(Compare, DomainT);
```

2.
```
domain_type lower() const;
```

3.
```
domain_type upper() const;
```

4.
```
interval_bounds bounds() const;
```

`continuous_interval` public static functions

1.
```
static continuous_interval open(const DomainT & lo, const DomainT & up);
```

2.
```
static continuous_interval right_open(const DomainT & lo, const DomainT & up);
```

3.
```
static continuous_interval left_open(const DomainT & lo, const DomainT & up);
```

4.
```
static continuous_interval closed(const DomainT & lo, const DomainT & up);
```

Struct template interval_traits<icl::continuous_interval< DomainT, Compare >>

boost::icl::interval_traits<icl::continuous_interval< DomainT, Compare >>

Synopsis

```
// In header: <boost/icl/continuous_interval.hpp>

template<typename DomainT, ICL_COMPARE Compare>
struct interval_traits<icl::continuous_interval< DomainT, Compare >> {
  // types
  typedef interval_traits                               type;
  typedef DomainT                                       domain_type;
  typedef icl::continuous_interval< DomainT, Compare > interval_type;

  // public member functions
  typedef ICL_COMPARE_DOMAIN(Compare, DomainT);

  // public static functions
  static interval_type construct(const domain_type &, const domain_type &);
  static domain_type lower(const interval_type &);
  static domain_type upper(const interval_type &);
};
```

Description

`interval_traits` public member functions

1.
```
typedef ICL_COMPARE_DOMAIN(Compare, DomainT);
```

`interval_traits` public static functions

1.
```
static interval_type construct(const domain_type & lo, const domain_type & up);
```

2.
```
static domain_type lower(const interval_type & inter_val);
```

3.
```
static domain_type upper(const interval_type & inter_val);
```

Struct template dynamic_interval_traits<boost::icl::continuous_interval< DomainT, Compare >>

boost::icl::dynamic_interval_traits<boost::icl::continuous_interval< DomainT, Compare >>

Synopsis

```
// In header: <boost/icl/continuous_interval.hpp>

template<typename DomainT, ICL_COMPARE Compare>
struct dynamic_interval_traits<boost::icl::continuous_interval< DomainT, Compare >> {
  // types
  typedef dynamic_interval_traits                          type;
  typedef boost::icl::continuous_interval< DomainT, Compare > interval_type;
  typedef DomainT                                          domain_type;

  // public member functions
  typedef ICL_COMPARE_DOMAIN(Compare, DomainT);

  // public static functions
  static interval_type
  construct(const domain_type, const domain_type, interval_bounds);
  static interval_type
  construct_bounded(const bounded_value< DomainT > &,
                    const bounded_value< DomainT > &);
};
```

Description

`dynamic_interval_traits` public member functions

1.
```
typedef ICL_COMPARE_DOMAIN(Compare, DomainT);
```

`dynamic_interval_traits` public static functions

1.
```
static interval_type
construct(const domain_type lo, const domain_type up, interval_bounds bounds);
```

2.
```
static interval_type
construct_bounded(const bounded_value< DomainT > & lo,
                  const bounded_value< DomainT > & up);
```

Struct template interval_bound_type<continuous_interval< DomainT, Compare >>

boost::icl::interval_bound_type<continuous_interval< DomainT, Compare >>

Synopsis

```
// In header: <boost/icl/continuous_interval.hpp>

template<typename DomainT, ICL_COMPARE Compare>
struct interval_bound_type<continuous_interval< DomainT, Compare >> {
  // types
  typedef interval_bound_type type;

  // public member functions
   BOOST_STATIC_CONSTANT(bound_type, value = interval_bounds::dynamic);
};
```

Description

`interval_bound_type` public member functions

1.
```
BOOST_STATIC_CONSTANT(bound_type, value = interval_bounds::dynamic);
```

Struct template is_continuous_interval<continuous_interval< DomainT, Compare >>

boost::icl::is_continuous_interval<continuous_interval< DomainT, Compare >>

Synopsis

```
// In header: <boost/icl/continuous_interval.hpp>

template<typename DomainT, ICL_COMPARE Compare>
struct is_continuous_interval<continuous_interval< DomainT, Compare >> {
  // types
  typedef is_continuous_interval< continuous_interval< DomainT, Compare > > type;

  // public member functions
   BOOST_STATIC_CONSTANT(bool, value = true);
};
```

Description

`is_continuous_interval` public member functions

1.
```
BOOST_STATIC_CONSTANT(bool, value = true);
```

Struct template type_to_string<icl::continuous_interval< DomainT, Compare >>

boost::icl::type_to_string<icl::continuous_interval< DomainT, Compare >>

Synopsis

```
// In header: <boost/icl/continuous_interval.hpp>

template<typename DomainT, ICL_COMPARE Compare>
struct type_to_string<icl::continuous_interval< DomainT, Compare >> {

  // public static functions
  static std::string apply();
};
```

Description

`type_to_string` public static functions

1.
```
static std::string apply();
```

Struct template value_size<icl::continuous_interval< DomainT >>

boost::icl::value_size<icl::continuous_interval< DomainT >>

Synopsis

```
// In header: <boost/icl/continuous_interval.hpp>

template<typename DomainT>
struct value_size<icl::continuous_interval< DomainT >> {

  // public static functions
  static std::size_t apply(const icl::continuous_interval< DomainT > &);
};
```

Description

`value_size` public static functions

1.
```
static std::size_t apply(const icl::continuous_interval< DomainT > &);
```

Header <boost/icl/discrete_interval.hpp>

```
namespace boost {
  namespace icl {
    template<typename DomainT, 
             ICL_COMPARE Compare = ICL_COMPARE_INSTANCE(ICL_COMPARE_DEFAULT, DomainT)> 
      class discrete_interval;

    template<typename DomainT, ICL_COMPARE Compare> 
      struct interval_traits<icl::discrete_interval< DomainT, Compare >>;
    template<typename DomainT, ICL_COMPARE Compare> 
      struct dynamic_interval_traits<boost::icl::discrete_interval< DomainT, Compare >>;
    template<typename DomainT, ICL_COMPARE Compare> 
      struct interval_bound_type<discrete_interval< DomainT, Compare >>;
    template<typename DomainT, ICL_COMPARE Compare> 
      struct is_discrete_interval<discrete_interval< DomainT, Compare >>;
    template<typename DomainT, ICL_COMPARE Compare> 
      struct type_to_string<icl::discrete_interval< DomainT, Compare >>;
    template<typename DomainT> 
      struct value_size<icl::discrete_interval< DomainT >>;
  }
}
```

Class template discrete_interval

boost::icl::discrete_interval

Synopsis

```
// In header: <boost/icl/discrete_interval.hpp>

template<typename DomainT, 
         ICL_COMPARE Compare = ICL_COMPARE_INSTANCE(ICL_COMPARE_DEFAULT, DomainT)> 
class discrete_interval {
public:
  // types
  typedef discrete_interval< DomainT, Compare > type;              
  typedef DomainT                               domain_type;        
  typedef bounded_value< DomainT >::type        bounded_domain_type;

  // construct/copy/destruct
  discrete_interval();
  explicit discrete_interval(const DomainT &);
  discrete_interval(const DomainT &, const DomainT &, 
                    interval_bounds = interval_bounds::right_open(), 
                    discrete_interval * = 0);

  // public member functions
  typedef ICL_COMPARE_DOMAIN(Compare, DomainT);
  domain_type lower() const;
  domain_type upper() const;
  interval_bounds bounds() const;

  // public static functions
  static discrete_interval open(const DomainT &, const DomainT &);
  static discrete_interval right_open(const DomainT &, const DomainT &);
  static discrete_interval left_open(const DomainT &, const DomainT &);
  static discrete_interval closed(const DomainT &, const DomainT &);
};
```

Description

`discrete_interval` public construct/copy/destruct

1.
```
discrete_interval();
```

   Default constructor; yields an empty interval `[0,0)`.

2.
```
explicit discrete_interval(const DomainT & val);
```

   Constructor for a closed singleton interval `[val,val]`

3.
```
discrete_interval(const DomainT & low, const DomainT & up,
                  interval_bounds bounds = interval_bounds::right_open(),
                  discrete_interval * = 0);
```

   Interval from `low` to `up` with bounds `bounds`

`discrete_interval` public member functions

1.
```
typedef ICL_COMPARE_DOMAIN(Compare, DomainT);
```

2.
```
domain_type lower() const;
```

3.
```
domain_type upper() const;
```

4.
```
interval_bounds bounds() const;
```

`discrete_interval` public static functions

1.
```
static discrete_interval open(const DomainT & lo, const DomainT & up);
```

2.
```
static discrete_interval right_open(const DomainT & lo, const DomainT & up);
```

3.
```
static discrete_interval left_open(const DomainT & lo, const DomainT & up);
```

4.
```
static discrete_interval closed(const DomainT & lo, const DomainT & up);
```

Struct template interval_traits<icl::discrete_interval< DomainT, Compare >>

boost::icl::interval_traits<icl::discrete_interval< DomainT, Compare >>

Synopsis

```
// In header: <boost/icl/discrete_interval.hpp>

template<typename DomainT, ICL_COMPARE Compare>
struct interval_traits<icl::discrete_interval< DomainT, Compare >> {
  // types
  typedef interval_traits                             type;
  typedef DomainT                                     domain_type;
  typedef icl::discrete_interval< DomainT, Compare > interval_type;

  // public member functions
  typedef ICL_COMPARE_DOMAIN(Compare, DomainT);

  // public static functions
  static interval_type construct(const domain_type &, const domain_type &);
  static domain_type lower(const interval_type &);
  static domain_type upper(const interval_type &);
};
```

Description

`interval_traits` public member functions

1.
```
typedef ICL_COMPARE_DOMAIN(Compare, DomainT);
```

`interval_traits` public static functions

1.
```
static interval_type construct(const domain_type & lo, const domain_type & up);
```

2.
```
static domain_type lower(const interval_type & inter_val);
```

3.
```
static domain_type upper(const interval_type & inter_val);
```

Struct template dynamic_interval_traits<boost::icl::discrete_interval< DomainT, Compare >>

boost::icl::dynamic_interval_traits<boost::icl::discrete_interval< DomainT, Compare >>

Synopsis

```
// In header: <boost/icl/discrete_interval.hpp>

template<typename DomainT, ICL_COMPARE Compare>
struct dynamic_interval_traits<boost::icl::discrete_interval< DomainT, Compare >> {
  // types
  typedef dynamic_interval_traits                          type;
  typedef boost::icl::discrete_interval< DomainT, Compare > interval_type;
  typedef DomainT                                          domain_type;

  // public member functions
  typedef ICL_COMPARE_DOMAIN(Compare, DomainT);

  // public static functions
  static interval_type
  construct(const domain_type &, const domain_type &, interval_bounds);
  static interval_type
  construct_bounded(const bounded_value< DomainT > &,
                    const bounded_value< DomainT > &);
};
```

Description

`dynamic_interval_traits` public member functions

1.
```
typedef ICL_COMPARE_DOMAIN(Compare, DomainT);
```

`dynamic_interval_traits` public static functions

1.
```
static interval_type
construct(const domain_type & lo, const domain_type & up,
          interval_bounds bounds);
```

2.
```
static interval_type
construct_bounded(const bounded_value< DomainT > & lo,
                  const bounded_value< DomainT > & up);
```

Struct template interval_bound_type<discrete_interval< DomainT, Compare >>

boost::icl::interval_bound_type<discrete_interval< DomainT, Compare >>

Synopsis

```
// In header: <boost/icl/discrete_interval.hpp>

template<typename DomainT, ICL_COMPARE Compare>
struct interval_bound_type<discrete_interval< DomainT, Compare >> {
  // types
  typedef interval_bound_type type;

  // public member functions
   BOOST_STATIC_CONSTANT(bound_type, value = interval_bounds::dynamic);
};
```

Description

`interval_bound_type` public member functions

1. ```
   BOOST_STATIC_CONSTANT(bound_type, value = interval_bounds::dynamic);
   ```

Struct template is_discrete_interval<discrete_interval< DomainT, Compare >>

boost::icl::is_discrete_interval<discrete_interval< DomainT, Compare >>

Synopsis

```
// In header: <boost/icl/discrete_interval.hpp>

template<typename DomainT, ICL_COMPARE Compare>
struct is_discrete_interval<discrete_interval< DomainT, Compare >> {
  // types
  typedef is_discrete_interval< discrete_interval< DomainT, Compare > > type;

  // public member functions
   BOOST_STATIC_CONSTANT(bool, value = is_discrete< DomainT >::value);
};
```

Description

`is_discrete_interval` public member functions

1. ```
   BOOST_STATIC_CONSTANT(bool, value = is_discrete< DomainT >::value);
   ```

Struct template type_to_string<icl::discrete_interval< DomainT, Compare >>

boost::icl::type_to_string<icl::discrete_interval< DomainT, Compare >>

Synopsis

```
// In header: <boost/icl/discrete_interval.hpp>

template<typename DomainT, ICL_COMPARE Compare>
struct type_to_string<icl::discrete_interval< DomainT, Compare >> {

  // public static functions
  static std::string apply();
};
```

Description

`type_to_string` public static functions

1. ```
   static std::string apply();
   ```

Struct template value_size<icl::discrete_interval< DomainT >>

boost::icl::value_size<icl::discrete_interval< DomainT >>

Synopsis

```
// In header: <boost/icl/discrete_interval.hpp>

template<typename DomainT>
struct value_size<icl::discrete_interval< DomainT >> {

  // public static functions
  static std::size_t apply(const icl::discrete_interval< DomainT > &);
};
```

Description

`value_size` public static functions

1. ```
   static std::size_t apply(const icl::discrete_interval< DomainT > &);
   ```

Header <boost/icl/dynamic_interval_traits.hpp>

```
namespace boost {
  namespace icl {
    template<typename DomainT> class bounded_value;

    template<typename Type> struct dynamic_interval_traits;
  }
}
```

Class template bounded_value

boost::icl::bounded_value

Synopsis

```
// In header: <boost/icl/dynamic_interval_traits.hpp>

template<typename DomainT>
class bounded_value {
public:
  // construct/copy/destruct
  bounded_value(const domain_type &, interval_bounds);

  // public member functions
  domain_type value() const;
  interval_bounds bound() const;
};
```

Description

`bounded_value` public construct/copy/destruct

1.
```
bounded_value(const domain_type & value, interval_bounds bound);
```

`bounded_value` public member functions

1.
```
domain_type value() const;
```

2.
```
interval_bounds bound() const;
```

Struct template dynamic_interval_traits

boost::icl::dynamic_interval_traits

Synopsis

```
// In header: <boost/icl/dynamic_interval_traits.hpp>

template<typename Type>
struct dynamic_interval_traits {
  // types
  typedef Type::domain_type    domain_type;
  typedef Type::domain_compare domain_compare;

  // public static functions
  static Type construct(const domain_type &, const domain_type &,
                        interval_bounds);
  static Type construct_bounded(const bounded_value< domain_type > &,
                                const bounded_value< domain_type > &);
};
```

Description

`dynamic_interval_traits` public static functions

1.
```
static Type construct(const domain_type & lo, const domain_type & up, 
                      interval_bounds bounds);
```

2.
```
static Type construct_bounded(const bounded_value< domain_type > & lo, 
                              const bounded_value< domain_type > & up);
```

Header <boost/icl/functors.hpp>

```
namespace boost {
  namespace icl {
    template<typename Type> struct identity_based_inplace_combine;
    template<typename Type> struct unit_element_based_inplace_combine;
    template<typename Type> struct inplace_identity;
    template<typename Type> struct inplace_erasure;
    template<typename Type> struct inplace_plus;
    template<typename Type> struct inplace_minus;
    template<typename Type> struct inplace_bit_add;
    template<typename Type> struct inplace_bit_subtract;
    template<typename Type> struct inplace_bit_and;
    template<typename Type> struct inplace_bit_xor;
    template<typename Type> struct inplace_et;
    template<typename Type> struct inplace_caret;
    template<typename Type> struct inplace_insert;
    template<typename Type> struct inplace_erase;
    template<typename Type> struct inplace_star;
    template<typename Type> struct inplace_slash;
    template<typename Type> struct inplace_max;
    template<typename Type> struct inplace_min;
    template<typename Type> struct inter_section;
    template<typename Functor> struct inverse;

    template<typename Type> struct inverse<icl::inplace_plus< Type >>;
    template<typename Type> struct inverse<icl::inplace_minus< Type >>;
    template<typename Type> struct inverse<icl::inplace_bit_add< Type >>;
    template<typename Type> struct inverse<icl::inplace_bit_subtract< Type >>;
    template<typename Type> struct inverse<icl::inplace_et< Type >>;
    template<typename Type> struct inverse<icl::inplace_caret< Type >>;
    template<typename Type> struct inverse<icl::inplace_bit_and< Type >>;
    template<typename Type> struct inverse<icl::inplace_bit_xor< Type >>;
    template<typename Type> struct inverse<icl::inplace_star< Type >>;
    template<typename Type> struct inverse<icl::inplace_slash< Type >>;
    template<typename Type> struct inverse<icl::inplace_max< Type >>;
    template<typename Type> struct inverse<icl::inplace_min< Type >>;
    template<typename Type> struct inverse<icl::inplace_identity< Type >>;
    template<typename Type> struct inverse<icl::inter_section< Type >>;

    template<typename Functor> struct is_negative;

    template<typename Type> struct is_negative<icl::inplace_minus< Type >>;
    template<typename Type> 
      struct is_negative<icl::inplace_bit_subtract< Type >>;

    template<typename Combiner> struct conversion;
    template<typename Combiner> struct version;
```

```
    template<> struct version<icl::inplace_minus< short >>;
    template<> struct version<icl::inplace_minus< int >>;
    template<> struct version<icl::inplace_minus< long >>;
    template<> struct version<icl::inplace_minus< long long >>;
    template<> struct version<icl::inplace_minus< float >>;
    template<> struct version<icl::inplace_minus< double >>;
    template<> struct version<icl::inplace_minus< long double >>;
    template<typename Type> struct version<icl::inplace_minus< Type >>;
  }
}
```

Struct template identity_based_inplace_combine

boost::icl::identity_based_inplace_combine

Synopsis

```
// In header: <boost/icl/functors.hpp>

template<typename Type>
struct identity_based_inplace_combine :
  public std::binary_function< Type &, const Type &, void >
{

  // public static functions
  static Type identity_element();
};
```

Description

`identity_based_inplace_combine` public static functions

1.
```
static Type identity_element();
```

Struct template unit_element_based_inplace_combine

boost::icl::unit_element_based_inplace_combine

Synopsis

```
// In header: <boost/icl/functors.hpp>

template<typename Type>
struct unit_element_based_inplace_combine :
  public std::binary_function< Type &, const Type &, void >
{

  // public static functions
  static Type identity_element();
};
```

Description

`unit_element_based_inplace_combine` public static functions

1.
```
static Type identity_element();
```

Struct template inplace_identity

boost::icl::inplace_identity

Synopsis

```
// In header: <boost/icl/functors.hpp>

template<typename Type>
struct inplace_identity :
  public boost::icl::identity_based_inplace_combine< Type >
{
  // types
  typedef inplace_identity< Type > type;

  // public member functions
  void operator()(Type &, const Type &) const;
};
```

Description

`inplace_identity` public member functions

1.
```
void operator()(Type &, const Type &) const;
```

Struct template inplace_erasure

boost::icl::inplace_erasure

Synopsis

```
// In header: <boost/icl/functors.hpp>

template<typename Type>
struct inplace_erasure :
  public boost::icl::identity_based_inplace_combine< Type >
{
  // types
  typedef inplace_erasure< Type >                type;
  typedef identity_based_inplace_combine< Type > base_type;

  // public member functions
  void operator()(Type &, const Type &) const;
};
```

Description

`inplace_erasure` public member functions

1.
```
void operator()(Type & object, const Type & operand) const;
```

Struct template inplace_plus

boost::icl::inplace_plus

Synopsis

```
// In header: <boost/icl/functors.hpp>

template<typename Type>
struct inplace_plus :
  public boost::icl::identity_based_inplace_combine< Type >
{
  // types
  typedef inplace_plus< Type > type;

  // public member functions
  void operator()(Type &, const Type &) const;

  // public static functions
  static void version(Type &);
};
```

Description

`inplace_plus` public member functions

1.
```
void operator()(Type & object, const Type & operand) const;
```

`inplace_plus` public static functions

1.
```
static void version(Type &);
```

Struct template inplace_minus

boost::icl::inplace_minus

Synopsis

```
// In header: <boost/icl/functors.hpp>

template<typename Type>
struct inplace_minus :
  public boost::icl::identity_based_inplace_combine< Type >
{
  // types
  typedef inplace_minus< Type > type;

  // public member functions
  void operator()(Type &, const Type &) const;
};
```

Description

`inplace_minus` public member functions

1.
```
void operator()(Type & object, const Type & operand) const;
```

Struct template inplace_bit_add

boost::icl::inplace_bit_add

Synopsis

```
// In header: <boost/icl/functors.hpp>

template<typename Type>
struct inplace_bit_add :
  public boost::icl::identity_based_inplace_combine< Type >
{
  // types
  typedef inplace_bit_add< Type > type;

  // public member functions
  void operator()(Type &, const Type &) const;

  // public static functions
  static void version(Type &);
};
```

Description

`inplace_bit_add` public member functions

1.
```
void operator()(Type & object, const Type & operand) const;
```

`inplace_bit_add` public static functions

1.
```
static void version(Type &);
```

Struct template inplace_bit_subtract

boost::icl::inplace_bit_subtract

Synopsis

```
// In header: <boost/icl/functors.hpp>

template<typename Type>
struct inplace_bit_subtract :
  public boost::icl::identity_based_inplace_combine< Type >
{
  // types
  typedef inplace_bit_subtract< Type > type;

  // public member functions
  void operator()(Type &, const Type &) const;
};
```

Description

`inplace_bit_subtract` public member functions

1.
```
void operator()(Type & object, const Type & operand) const;
```

Struct template inplace_bit_and

boost::icl::inplace_bit_and

Synopsis

```
// In header: <boost/icl/functors.hpp>

template<typename Type>
struct inplace_bit_and :
  public boost::icl::identity_based_inplace_combine< Type >
{
  // types
  typedef inplace_bit_and< Type > type;

  // public member functions
  void operator()(Type &, const Type &) const;
};
```

Description

`inplace_bit_and` public member functions

1.
```
void operator()(Type & object, const Type & operand) const;
```

Struct template inplace_bit_xor

boost::icl::inplace_bit_xor

Synopsis

```
// In header: <boost/icl/functors.hpp>

template<typename Type>
struct inplace_bit_xor :
  public boost::icl::identity_based_inplace_combine< Type >
{
  // types
  typedef inplace_bit_xor< Type > type;

  // public member functions
  void operator()(Type &, const Type &) const;
};
```

Description

`inplace_bit_xor` public member functions

1.
```
void operator()(Type & object, const Type & operand) const;
```

Struct template inplace_et

boost::icl::inplace_et

Synopsis

```
// In header: <boost/icl/functors.hpp>

template<typename Type>
struct inplace_et : public boost::icl::identity_based_inplace_combine< Type > {
  // types
  typedef inplace_et< Type > type;

  // public member functions
  void operator()(Type &, const Type &) const;
};
```

Description

`inplace_et` public member functions

1.
```
void operator()(Type & object, const Type & operand) const;
```

Struct template inplace_caret

boost::icl::inplace_caret

Synopsis

```
// In header: <boost/icl/functors.hpp>

template<typename Type>
struct inplace_caret :
  public boost::icl::identity_based_inplace_combine< Type >
{
  // types
  typedef inplace_caret< Type > type;

  // public member functions
  void operator()(Type &, const Type &) const;
};
```

Description

`inplace_caret` public member functions

1.
```
void operator()(Type & object, const Type & operand) const;
```

Struct template inplace_insert

boost::icl::inplace_insert

Synopsis

```
// In header: <boost/icl/functors.hpp>

template<typename Type>
struct inplace_insert :
  public boost::icl::identity_based_inplace_combine< Type >
{
  // types
  typedef inplace_insert< Type > type;

  // public member functions
  void operator()(Type &, const Type &) const;
};
```

Description

`inplace_insert` public member functions

1.
```
void operator()(Type & object, const Type & operand) const;
```

Struct template inplace_erase

boost::icl::inplace_erase

Synopsis

```
// In header: <boost/icl/functors.hpp>

template<typename Type>
struct inplace_erase :
  public boost::icl::identity_based_inplace_combine< Type >
{
  // types
  typedef inplace_erase< Type > type;

  // public member functions
  void operator()(Type &, const Type &) const;
};
```

Description

`inplace_erase` public member functions

1.
```
void operator()(Type & object, const Type & operand) const;
```

Struct template inplace_star

boost::icl::inplace_star

Synopsis

```
// In header: <boost/icl/functors.hpp>

template<typename Type>
struct inplace_star :
  public boost::icl::identity_based_inplace_combine< Type >
{
  // types
  typedef inplace_star< Type > type;

  // public member functions
  void operator()(Type &, const Type &) const;
};
```

Description

`inplace_star` public member functions

1.
```
void operator()(Type & object, const Type & operand) const;
```

Struct template inplace_slash

boost::icl::inplace_slash

Synopsis

```
// In header: <boost/icl/functors.hpp>

template<typename Type>
struct inplace_slash :
  public boost::icl::identity_based_inplace_combine< Type >
{
  // types
  typedef inplace_slash< Type > type;

  // public member functions
  void operator()(Type &, const Type &) const;
};
```

Description

`inplace_slash` public member functions

1.
```
void operator()(Type & object, const Type & operand) const;
```

Struct template inplace_max

boost::icl::inplace_max

Synopsis

```
// In header: <boost/icl/functors.hpp>

template<typename Type>
struct inplace_max :
  public boost::icl::identity_based_inplace_combine< Type >
{
  // types
  typedef inplace_max< Type > type;

  // public member functions
  void operator()(Type &, const Type &) const;
};
```

Description

`inplace_max` public member functions

1.
```
void operator()(Type & object, const Type & operand) const;
```

Struct template inplace_min

boost::icl::inplace_min

Synopsis

```
// In header: <boost/icl/functors.hpp>

template<typename Type>
struct inplace_min :
  public boost::icl::identity_based_inplace_combine< Type >
{
  // types
  typedef inplace_min< Type > type;

  // public member functions
  void operator()(Type &, const Type &) const;
};
```

Description

`inplace_min` public member functions

1.
```
void operator()(Type & object, const Type & operand) const;
```

Struct template inter_section

boost::icl::inter_section

Synopsis

```
// In header: <boost/icl/functors.hpp>

template<typename Type>
struct inter_section :
  public boost::icl::identity_based_inplace_combine< Type >
{
  // types
  typedef boost::mpl::if_< has_set_semantics< Type >, icl::inplace_et< Type >, icl::in↵
place_plus< Type > >::type type;

  // public member functions
  void operator()(Type &, const Type &) const;
};
```

Description

`inter_section` public member functions

1.
```
void operator()(Type & object, const Type & operand) const;
```

Struct template inverse

boost::icl::inverse

Synopsis

```
// In header: <boost/icl/functors.hpp>

template<typename Functor>
struct inverse {
  // types
  typedef remove_reference< typename Functor::first_argument_type >::type argument_type;
  typedef icl::inplace_erasure< argument_type >                                type;
};
```

Struct template inverse<icl::inplace_plus< Type >>

boost::icl::inverse<icl::inplace_plus< Type >>

Synopsis

```
// In header: <boost/icl/functors.hpp>

template<typename Type>
struct inverse<icl::inplace_plus< Type >> {
  // types
  typedef icl::inplace_minus< Type > type;
};
```

Struct template inverse<icl::inplace_minus< Type >>

boost::icl::inverse<icl::inplace_minus< Type >>

Synopsis

```
// In header: <boost/icl/functors.hpp>

template<typename Type>
struct inverse<icl::inplace_minus< Type >> {
  // types
  typedef icl::inplace_plus< Type > type;
};
```

Struct template inverse<icl::inplace_bit_add< Type >>

boost::icl::inverse<icl::inplace_bit_add< Type >>

Synopsis

```
// In header: <boost/icl/functors.hpp>

template<typename Type>
struct inverse<icl::inplace_bit_add< Type >> {
  // types
  typedef icl::inplace_bit_subtract< Type > type;
};
```

Struct template inverse<icl::inplace_bit_subtract< Type >>

boost::icl::inverse<icl::inplace_bit_subtract< Type >>

Synopsis

```
// In header: <boost/icl/functors.hpp>

template<typename Type>
struct inverse<icl::inplace_bit_subtract< Type >> {
  // types
  typedef icl::inplace_bit_add< Type > type;
};
```

Struct template inverse<icl::inplace_et< Type >>

boost::icl::inverse<icl::inplace_et< Type >>

Synopsis

```
// In header: <boost/icl/functors.hpp>

template<typename Type>
struct inverse<icl::inplace_et< Type >> {
  // types
  typedef icl::inplace_caret< Type > type;
};
```

Struct template inverse<icl::inplace_caret< Type >>

boost::icl::inverse<icl::inplace_caret< Type >>

Synopsis

```
// In header: <boost/icl/functors.hpp>

template<typename Type>
struct inverse<icl::inplace_caret< Type >> {
  // types
  typedef icl::inplace_et< Type > type;
};
```

Struct template inverse<icl::inplace_bit_and< Type >>

boost::icl::inverse<icl::inplace_bit_and< Type >>

Synopsis

```
// In header: <boost/icl/functors.hpp>

template<typename Type>
struct inverse<icl::inplace_bit_and< Type >> {
  // types
  typedef icl::inplace_bit_xor< Type > type;
};
```

Struct template inverse<icl::inplace_bit_xor< Type >>

boost::icl::inverse<icl::inplace_bit_xor< Type >>

Synopsis

```
// In header: <boost/icl/functors.hpp>

template<typename Type>
struct inverse<icl::inplace_bit_xor< Type >> {
  // types
  typedef icl::inplace_bit_and< Type > type;
};
```

Struct template inverse<icl::inplace_star< Type >>

boost::icl::inverse<icl::inplace_star< Type >>

Synopsis

```
// In header: <boost/icl/functors.hpp>

template<typename Type>
struct inverse<icl::inplace_star< Type >> {
  // types
  typedef icl::inplace_slash< Type > type;
};
```

Struct template inverse<icl::inplace_slash< Type >>

boost::icl::inverse<icl::inplace_slash< Type >>

Synopsis

```
// In header: <boost/icl/functors.hpp>

template<typename Type>
struct inverse<icl::inplace_slash< Type >> {
  // types
  typedef icl::inplace_star< Type > type;
};
```

Struct template inverse<icl::inplace_max< Type >>

boost::icl::inverse<icl::inplace_max< Type >>

Synopsis

```
// In header: <boost/icl/functors.hpp>

template<typename Type>
struct inverse<icl::inplace_max< Type >> {
  // types
  typedef icl::inplace_min< Type > type;
};
```

Struct template inverse<icl::inplace_min< Type >>

boost::icl::inverse<icl::inplace_min< Type >>

Synopsis

```
// In header: <boost/icl/functors.hpp>

template<typename Type>
struct inverse<icl::inplace_min< Type >> {
  // types
  typedef icl::inplace_max< Type > type;
};
```

Struct template inverse<icl::inplace_identity< Type >>

boost::icl::inverse<icl::inplace_identity< Type >>

Synopsis

```
// In header: <boost/icl/functors.hpp>

template<typename Type>
struct inverse<icl::inplace_identity< Type >> {
  // types
  typedef icl::inplace_erasure< Type > type;
};
```

Struct template inverse<icl::inter_section< Type >>

boost::icl::inverse<icl::inter_section< Type >>

Synopsis

```
// In header: <boost/icl/functors.hpp>

template<typename Type> 
struct inverse<icl::inter_section< Type >> : public boost::icl::identity_based_inplace_com↵
bine< Type > {
  // types
  typedef boost::mpl::if_< has_set_semantics< Type >, icl::inplace_caret< Type >, icl::in↵
place_minus< Type > >::type type;

  // public member functions
  void operator()(Type &, const Type &) const;
};
```

Description

`inverse` public member functions

1.
```
void operator()(Type & object, const Type & operand) const;
```

Struct template is_negative

boost::icl::is_negative

Synopsis

```
// In header: <boost/icl/functors.hpp>

template<typename Functor> 
struct is_negative {
  // types
  typedef is_negative< Functor > type;

  // public member functions
   BOOST_STATIC_CONSTANT(bool, value = false);
};
```

Description

`is_negative` public member functions

1.
```
 BOOST_STATIC_CONSTANT(bool, value = false);
```

Struct template is_negative<icl::inplace_minus< Type >>

boost::icl::is_negative<icl::inplace_minus< Type >>

Synopsis

```
// In header: <boost/icl/functors.hpp>

template<typename Type>
struct is_negative<icl::inplace_minus< Type >> {
  // types
  typedef is_negative type;

  // public member functions
   BOOST_STATIC_CONSTANT(bool, value = true);
};
```

Description

`is_negative` public member functions

1. ```
   BOOST_STATIC_CONSTANT(bool, value = true);
   ```

Struct template is_negative<icl::inplace_bit_subtract< Type >>

boost::icl::is_negative<icl::inplace_bit_subtract< Type >>

Synopsis

```
// In header: <boost/icl/functors.hpp>

template<typename Type>
struct is_negative<icl::inplace_bit_subtract< Type >> {
  // types
  typedef is_negative type;

  // public member functions
   BOOST_STATIC_CONSTANT(bool, value = true);
};
```

Description

`is_negative` public member functions

1. ```
   BOOST_STATIC_CONSTANT(bool, value = true);
   ```

Struct template conversion

boost::icl::conversion

Synopsis

```
// In header: <boost/icl/functors.hpp>

template<typename Combiner>
struct conversion {
  // types
  typedef conversion< Combiner >                                             ↵
          type;
  typedef remove_const< typename remove_reference< typename Combiner::first_argu↵
ment_type >::type >::type argument_type;

  // public static functions
  static argument_type proversion(const argument_type &);
  static argument_type inversion(const argument_type &);
};
```

Description

`conversion` public static functions

1.
```
static argument_type proversion(const argument_type & value);
```

2.
```
static argument_type inversion(const argument_type & value);
```

Struct template version

boost::icl::version

Synopsis

```
// In header: <boost/icl/functors.hpp>

template<typename Combiner>
struct version : public boost::icl::conversion< Combiner > {
  // types
  typedef version< Combiner >      type;
  typedef conversion< Combiner >   base_type;
  typedef base_type::argument_type argument_type;

  // public member functions
  argument_type operator()(const argument_type &);
};
```

Description

`version` public member functions

1.
```
argument_type operator()(const argument_type & value);
```

Struct version<icl::inplace_minus< short >>

boost::icl::version<icl::inplace_minus< short >>

Synopsis

```
// In header: <boost/icl/functors.hpp>


struct version<icl::inplace_minus< short >> {

  // public member functions
  short operator()(short);
};
```

Description

`version` public member functions

1.
```
short operator()(short val);
```

Struct version<icl::inplace_minus< int >>

boost::icl::version<icl::inplace_minus< int >>

Synopsis

```
// In header: <boost/icl/functors.hpp>


struct version<icl::inplace_minus< int >> {

  // public member functions
  int operator()(int);
};
```

Description

`version` public member functions

1.
```
int operator()(int val);
```

Struct version<icl::inplace_minus< long >>

boost::icl::version<icl::inplace_minus< long >>

Synopsis

```
// In header: <boost/icl/functors.hpp>


struct version<icl::inplace_minus< long >> {

  // public member functions
  long operator()(long);
};
```

Description

`version` public member functions

1. ```
   long operator()(long val);
   ```

Struct version<icl::inplace_minus< long long >>

boost::icl::version<icl::inplace_minus< long long >>

Synopsis

```
// In header: <boost/icl/functors.hpp>


struct version<icl::inplace_minus< long long >> {

  // public member functions
  long long operator()(long long);
};
```

Description

`version` public member functions

1. ```
   long long operator()(long long val);
   ```

Struct version<icl::inplace_minus< float >>

boost::icl::version<icl::inplace_minus< float >>

Synopsis

```
// In header: <boost/icl/functors.hpp>


struct version<icl::inplace_minus< float >> {

  // public member functions
  float operator()(float);
};
```

Description

`version` public member functions

1.
```
float operator()(float val);
```

Struct version<icl::inplace_minus< double >>

boost::icl::version<icl::inplace_minus< double >>

Synopsis

```
// In header: <boost/icl/functors.hpp>


struct version<icl::inplace_minus< double >> {

  // public member functions
  double operator()(double);
};
```

Description

`version` public member functions

1.
```
double operator()(double val);
```

Struct version<icl::inplace_minus< long double >>

boost::icl::version<icl::inplace_minus< long double >>

Synopsis

```
// In header: <boost/icl/functors.hpp>


struct version<icl::inplace_minus< long double >> {

  // public member functions
  long double operator()(long double);
};
```

Description

`version` public member functions

1.
```
long double operator()(long double val);
```

Struct template version<icl::inplace_minus< Type >>

boost::icl::version<icl::inplace_minus< Type >>

Synopsis

```
// In header: <boost/icl/functors.hpp>

template<typename Type>
struct version<icl::inplace_minus< Type >> : public boost::icl::conversion< icl::in↵
place_minus< Type > > {
  // types
  typedef version< icl::inplace_minus< Type > >    type;
  typedef conversion< icl::inplace_minus< Type > > base_type;
  typedef base_type::argument_type                 argument_type;

  // public member functions
  Type operator()(const Type &);
};
```

Description

`version` public member functions

1.
```
Type operator()(const Type & value);
```

Header <boost/icl/gregorian.hpp>

```
namespace boost {
  namespace icl {
    template<> struct is_discrete<boost::gregorian::date>;
    template<> struct identity_element<boost::gregorian::date_duration>;
    template<> struct has_difference<boost::gregorian::date>;
    template<> struct difference_type_of<boost::gregorian::date>;
    template<> struct size_type_of<boost::gregorian::date>;
    template<> struct is_discrete<boost::gregorian::date_duration>;
    template<> struct has_difference<boost::gregorian::date_duration>;
    template<> struct size_type_of<boost::gregorian::date_duration>;
    boost::gregorian::date operator++(boost::gregorian::date & x);
    boost::gregorian::date operator--(boost::gregorian::date & x);
    boost::gregorian::date_duration
    operator++(boost::gregorian::date_duration & x);
    boost::gregorian::date_duration
    operator--(boost::gregorian::date_duration & x);
  }
}
```

Struct is_discrete<boost::gregorian::date>

boost::icl::is_discrete<boost::gregorian::date>

Synopsis

```
// In header: <boost/icl/gregorian.hpp>


struct is_discrete<boost::gregorian::date> {
  // types
  typedef is_discrete type;

  // public member functions
   BOOST_STATIC_CONSTANT(bool, value = true);
};
```

Description

`is_discrete` public member functions

1.
```
BOOST_STATIC_CONSTANT(bool, value = true);
```

Struct identity_element<boost::gregorian::date_duration>

boost::icl::identity_element<boost::gregorian::date_duration>

Synopsis

```
// In header: <boost/icl/gregorian.hpp>


struct identity_element<boost::gregorian::date_duration> {

  // public static functions
  static boost::gregorian::date_duration value();
};
```

Description

`identity_element` public static functions

1.
```
static boost::gregorian::date_duration value();
```

Struct has_difference<boost::gregorian::date>

boost::icl::has_difference<boost::gregorian::date>

Synopsis

```
// In header: <boost/icl/gregorian.hpp>


struct has_difference<boost::gregorian::date> {
  // types
  typedef has_difference type;

  // public member functions
   BOOST_STATIC_CONSTANT(bool, value = true);
};
```

Description

`has_difference` public member functions

1.
```
BOOST_STATIC_CONSTANT(bool, value = true);
```

Struct difference_type_of<boost::gregorian::date>

boost::icl::difference_type_of<boost::gregorian::date>

Synopsis

```
// In header: <boost/icl/gregorian.hpp>


struct difference_type_of<boost::gregorian::date> {
  // types
  typedef boost::gregorian::date_duration type;
};
```

Struct size_type_of<boost::gregorian::date>

boost::icl::size_type_of<boost::gregorian::date>

Synopsis

```
// In header: <boost/icl/gregorian.hpp>


struct size_type_of<boost::gregorian::date> {
  // types
  typedef boost::gregorian::date_duration type;
};
```

Struct is_discrete<boost::gregorian::date_duration>

boost::icl::is_discrete<boost::gregorian::date_duration>

Synopsis

```
// In header: <boost/icl/gregorian.hpp>


struct is_discrete<boost::gregorian::date_duration> {
  // types
  typedef is_discrete type;

  // public member functions
   BOOST_STATIC_CONSTANT(bool, value = true);
};
```

Description

`is_discrete` public member functions

1.
```
BOOST_STATIC_CONSTANT(bool, value = true);
```

Struct has_difference<boost::gregorian::date_duration>

boost::icl::has_difference<boost::gregorian::date_duration>

Synopsis

```
// In header: <boost/icl/gregorian.hpp>


struct has_difference<boost::gregorian::date_duration> {
  // types
  typedef has_difference type;

  // public member functions
   BOOST_STATIC_CONSTANT(bool, value = true);
};
```

Description

`has_difference` public member functions

1.
```
BOOST_STATIC_CONSTANT(bool, value = true);
```

Struct size_type_of<boost::gregorian::date_duration>

boost::icl::size_type_of<boost::gregorian::date_duration>

Synopsis

```
// In header: <boost/icl/gregorian.hpp>


struct size_type_of<boost::gregorian::date_duration> {
  // types
  typedef boost::gregorian::date_duration type;
};
```

Header <boost/icl/impl_config.hpp>

```


ICL_IMPL_SPACE
```

Macro ICL_IMPL_SPACE

ICL_IMPL_SPACE

Synopsis

```
// In header: <boost/icl/impl_config.hpp>

ICL_IMPL_SPACE
```

Header <boost/icl/interval.hpp>

```
namespace boost {
  namespace icl {
    template<typename IntervalT, bool IsDiscrete, bound_type PretendedBounds,
             bound_type RepresentedBounds>
      struct static_interval;
    template<typename DomainT,
             ICL_COMPARE Compare = ICL_COMPARE_INSTANCE(ICL_COMPARE_DEFAULT, DomainT)>
      struct interval;

    template<typename IntervalT, bound_type PretendedBounds,
             bound_type RepresentedBounds>
      struct static_interval<IntervalT, true, PretendedBounds, RepresentedBounds>;
    template<typename IntervalT, bound_type PretendedBounds,
             bound_type RepresentedBounds>
      struct static_interval<IntervalT, false, PretendedBounds, RepresentedBounds>;
  }
}
```

Struct template static_interval

boost::icl::static_interval

Synopsis

```
// In header: <boost/icl/interval.hpp>

template<typename IntervalT, bool IsDiscrete, bound_type PretendedBounds,
         bound_type RepresentedBounds>
struct static_interval {
};
```

Struct template interval

boost::icl::interval

Synopsis

```
// In header: <boost/icl/interval.hpp>

template<typename DomainT,
         ICL_COMPARE Compare = ICL_COMPARE_INSTANCE(ICL_COMPARE_DEFAULT, DomainT)>
struct interval {
  // types
  typedef interval_type_default< DomainT, Compare >::type interval_type;
  typedef interval_type                                   type;

  // public static functions
  static interval_type right_open(const DomainT &, const DomainT &);
  static interval_type left_open(const DomainT &, const DomainT &);
  static interval_type open(const DomainT &, const DomainT &);
  static interval_type closed(const DomainT &, const DomainT &);
  static interval_type construct(const DomainT &, const DomainT &);
};
```

Description

`interval` public static functions

1.
```
static interval_type right_open(const DomainT & low, const DomainT & up);
```

2.
```
static interval_type left_open(const DomainT & low, const DomainT & up);
```

3.
```
static interval_type open(const DomainT & low, const DomainT & up);
```

4.
```
static interval_type closed(const DomainT & low, const DomainT & up);
```

5.
```
static interval_type construct(const DomainT & low, const DomainT & up);
```

Struct template static_interval<IntervalT, true, PretendedBounds, RepresentedBounds>

boost::icl::static_interval<IntervalT, true, PretendedBounds, RepresentedBounds>

Synopsis

```
// In header: <boost/icl/interval.hpp>

template<typename IntervalT, bound_type PretendedBounds,
         bound_type RepresentedBounds>
struct static_interval<IntervalT, true, PretendedBounds, RepresentedBounds> {
  // types
  typedef interval_traits< IntervalT >::domain_type domain_type;

  // public static functions
  static IntervalT construct(const domain_type &, const domain_type &);
};
```

Description

`static_interval` public static functions

1.
```
static IntervalT construct(const domain_type & low, const domain_type & up);
```

Struct template static_interval<IntervalT, false, PretendedBounds, RepresentedBounds>

boost::icl::static_interval<IntervalT, false, PretendedBounds, RepresentedBounds>

Synopsis

```
// In header: <boost/icl/interval.hpp>

template<typename IntervalT, bound_type PretendedBounds,
         bound_type RepresentedBounds>
struct static_interval<IntervalT, false, PretendedBounds, RepresentedBounds> {
  // types
  typedef interval_traits< IntervalT >::domain_type domain_type;

  // public static functions
  static IntervalT construct(const domain_type &, const domain_type &);
};
```

Description

`static_interval` public static functions

1.
```
static IntervalT construct(const domain_type & low, const domain_type & up);
```

Header <boost/icl/interval_base_map.hpp>

```
namespace boost {
  namespace icl {
    template<typename DomainT, typename CodomainT> struct mapping_pair;

    template<typename SubType, typename DomainT, typename CodomainT, 
             typename Traits = icl::partial_absorber, 
             ICL_COMPARE Compare = ICL_COMPARE_INSTANCE(ICL_COMPARE_DEFAULT, DomainT), 
             ICL_COMBINE Combine = ICL_COMBINE_INSTANCE(icl::inplace_plus, CodomainT), 
             ICL_SECTION Section = ICL_SECTION_INSTANCE(icl::inter_section, CodomainT), 
           ICL_INTERVAL(ICL_COMPARE) Interval = ICL_INTERVAL_INSTANCE(ICL_INTERVAL_DEFAULT, Do↵
mainT, Compare), 
             ICL_ALLOC Alloc = std::allocator> 
      class interval_base_map;

    template<typename SubType, typename DomainT, typename CodomainT, 
             typename Traits, ICL_COMPARE Compare, ICL_COMBINE Combine, 
             ICL_SECTION Section, ICL_INTERVAL(ICL_COMPARE) Interval, 
             ICL_ALLOC Alloc> 
      struct is_map<icl::interval_base_map< SubType, DomainT, CodomainT, Traits, Compare, Com↵
bine, Section, Interval, Alloc >>;
    template<typename SubType, typename DomainT, typename CodomainT, 
             typename Traits, ICL_COMPARE Compare, ICL_COMBINE Combine, 
             ICL_SECTION Section, ICL_INTERVAL(ICL_COMPARE) Interval, 
             ICL_ALLOC Alloc> 
     struct has_inverse<icl::interval_base_map< SubType, DomainT, CodomainT, Traits, Compare, Com↵
bine, Section, Interval, Alloc >>;
    template<typename SubType, typename DomainT, typename CodomainT, 
             typename Traits, ICL_COMPARE Compare, ICL_COMBINE Combine, 
             ICL_SECTION Section, ICL_INTERVAL(ICL_COMPARE) Interval, 
             ICL_ALLOC Alloc> 
     struct is_interval_container<icl::interval_base_map< SubType, DomainT, CodomainT, Traits, Com↵
pare, Combine, Section, Interval, Alloc >>;
    template<typename SubType, typename DomainT, typename CodomainT, 
             typename Traits, ICL_COMPARE Compare, ICL_COMBINE Combine, 
             ICL_SECTION Section, ICL_INTERVAL(ICL_COMPARE) Interval, 
             ICL_ALLOC Alloc> 
     struct absorbs_identities<icl::interval_base_map< SubType, DomainT, CodomainT, Traits, Com↵
pare, Combine, Section, Interval, Alloc >>;
    template<typename SubType, typename DomainT, typename CodomainT, 
             typename Traits, ICL_COMPARE Compare, ICL_COMBINE Combine, 
             ICL_SECTION Section, ICL_INTERVAL(ICL_COMPARE) Interval, 
             ICL_ALLOC Alloc> 
     struct is_total<icl::interval_base_map< SubType, DomainT, CodomainT, Traits, Compare, Com↵
bine, Section, Interval, Alloc >>;
  }
}
```

Struct template mapping_pair

boost::icl::mapping_pair

Synopsis

```
// In header: <boost/icl/interval_base_map.hpp>

template<typename DomainT, typename CodomainT>
struct mapping_pair {
  // construct/copy/destruct
  mapping_pair();
  mapping_pair(const DomainT &, const CodomainT &);
  mapping_pair(const std::pair< DomainT, CodomainT > &);

  // public data members
  DomainT key;
  CodomainT data;
};
```

Description

`mapping_pair` public construct/copy/destruct

1. ```
   mapping_pair();
   ```

2. ```
   mapping_pair(const DomainT & key_value, const CodomainT & data_value);
   ```

3. ```
   mapping_pair(const std::pair< DomainT, CodomainT > & std_pair);
   ```

Class template interval_base_map

boost::icl::interval_base_map — Implements a map as a map of intervals (base class)
   ```

Synopsis

```
// In header: <boost/icl/interval_base_map.hpp>

template<typename SubType, typename DomainT, typename CodomainT,
         typename Traits = icl::partial_absorber,
         ICL_COMPARE Compare = ICL_COMPARE_INSTANCE(ICL_COMPARE_DEFAULT, DomainT),
         ICL_COMBINE Combine = ICL_COMBINE_INSTANCE(icl::inplace_plus, CodomainT),
         ICL_SECTION Section = ICL_SECTION_INSTANCE(icl::inter_section, CodomainT),
         ICL_INTERVAL(ICL_COMPARE) Interval = ICL_INTERVAL_INSTANCE(ICL_INTERVAL_DEFAULT, Do↵
mainT, Compare),
         ICL_ALLOC Alloc = std::allocator>
class interval_base_map {
public:
  // types
  typedef interval_base_map< SubType, DomainT, CodomainT, Traits, Compare, Combine, Section, In↵
terval, Alloc >                 type;
  typedef SubType                                                                                ↵
                                sub_type;                          // The designated de↵
rived or sub_type of this base class.
  typedef type                                                                                   ↵
                              overloadable_type;              // Auxilliary type for overloadresol↵
ution.
  typedef Traits                                                                                 ↵
                                traits;                            // Traits of an itl map.
  typedef icl::map< DomainT, CodomainT, Traits, Compare, Combine, Section, Alloc >               ↵
                               atomized_type;                   // The atomized type representing ↵
the corresponding container of elements.
  typedef DomainT                                                                                ↵
                               domain_type;                     // Domain type (type of the keys) ↵
of the map.
  typedef boost::call_traits< DomainT >::param_type                                              ↵
                                domain_param;
  typedef CodomainT                                                                              ↵
                               codomain_type;                   // Domain type (type of the keys) ↵
of the map.
  typedef mapping_pair< domain_type, codomain_type >                                             ↵
                                domain_mapping_type;              // Auxiliary type to help the com↵
piler resolve ambiguities when using std::make_pair.
  typedef domain_mapping_type                                                                    ↵
                               element_type;                    // Conceptual is a map a set of ele↵
ments of type element_type.
  typedef std::pair< interval_type, CodomainT >                                                  ↵
                               interval_mapping_type;           // Auxiliary type for overload res↵
olution.
  typedef std::pair< interval_type, CodomainT >                                                  ↵
                               segment_type;                    // Type of an interval containers ↵
segment, that is spanned by an interval.
  typedef difference_type_of< domain_type >::type                                                ↵
                               difference_type;                 // The difference type of an inter↵
val which is sometimes different form the domain_type.
  typedef size_type_of< domain_type >::type                                                      ↵
                                size_type;                         // The size type of an interval ↵
which is mostly std::size_t.
  typedef inverse< codomain_combine >::type                                                      ↵
                              inverse_codomain_combine;        // Inverse Combine functor for codo↵
main value aggregation.
  typedef mpl::if_< has_set_semantics< codomain_type >, ICL_SECTION_CODOMAIN(Section, Codo↵
mainT), codomain_combine >::type codomain_intersect;             // Intersection functor for ↵
codomain values.
  typedef inverse< codomain_intersect >::type                                                    ↵
                              inverse_codomain_intersect;      // Inverse Combine functor for codo↵
```

```
main value intersection.
  typedef exclusive_less_than< interval_type >                                  ↵
                          interval_compare;             // Comparison functor for intervals ↵
which are keys as well.
  typedef exclusive_less_than< interval_type >                                  ↵
                            key_compare;                  // Comparison functor for keys.
  typedef Alloc< std::pair< const interval_type, codomain_type > >              ↵
                            allocator_type;               // The allocator type of the set.
  typedef ICL_IMPL_SPACE::map< interval_type, codomain_type, key_compare, allocator_type >    ↵
                           ImplMapT;                    // Container type for the implement↵
ation.
  typedef ImplMapT::key_type                                                    ↵
                           key_type;                    // key type of the implementing con↵
tainer
  typedef ImplMapT::value_type                                                  ↵
                           value_type;                   // value type of the implementing ↵
container
  typedef ImplMapT::value_type::second_type                                     ↵
                            data_type;                    // data type of the implementing ↵
container
  typedef ImplMapT::pointer                                                     ↵
                            pointer;                      // pointer type
  typedef ImplMapT::const_pointer                                               ↵
                            const_pointer;                // const pointer type
  typedef ImplMapT::reference                                                   ↵
                            reference;                    // reference type
  typedef ImplMapT::const_reference                                             ↵
                            const_reference;              // const reference type
  typedef ImplMapT::iterator                                                    ↵
                            iterator;                     // iterator for iteration over in↵
tervals
  typedef ImplMapT::const_iterator                                              ↵
                            const_iterator;               // const_iterator for iteration ↵
over intervals
  typedef ImplMapT::reverse_iterator                                            ↵
                           reverse_iterator;             // iterator for reverse iteration ↵
over intervals
  typedef ImplMapT::const_reverse_iterator                                      ↵
                            const_reverse_iterator;       // const_iterator for iteration ↵
over intervals
  typedef boost::icl::element_iterator< iterator >                              ↵
                           element_iterator;             // element iterator: Depreciated, ↵
see documentation.
  typedef boost::icl::element_iterator< const_iterator >                        ↵
                           element_const_iterator;       // const element iterator: Depreci↵
ated, see documentation.
  typedef boost::icl::element_iterator< reverse_iterator >                      ↵
                           element_reverse_iterator;     // element reverse iterator: Depre↵
ciated, see documentation.
  typedef boost::icl::element_iterator< const_reverse_iterator >                ↵
                          element_const_reverse_iterator;  // element const reverse iterator: ↵
Depreciated, see documentation.
  typedef on_absorbtion< type, codomain_combine, Traits::absorbs_identities >::type          ↵
                            on_codomain_absorbtion;

  // member classes/structs/unions
  template<typename Type, bool has_set_semantics>
  struct on_codomain_model {
  };
  template<typename Type>
  struct on_codomain_model<Type, false> {
    // types
    typedef Type::interval_type    interval_type;
```

```
  typedef Type::codomain_type    codomain_type;
  typedef Type::segment_type     segment_type;
  typedef Type::codomain_combine codomain_combine;

  // public static functions
  static void add(Type &, interval_type &, const codomain_type &,
                  const codomain_type &);
};
template<typename Type>
struct on_codomain_model<Type, true> {
  // types
  typedef Type::interval_type                interval_type;
  typedef Type::codomain_type                codomain_type;
  typedef Type::segment_type                 segment_type;
  typedef Type::codomain_combine             codomain_combine;
  typedef Type::inverse_codomain_intersect inverse_codomain_intersect;

  // public static functions
  static void add(Type &, interval_type &, const codomain_type &,
                  const codomain_type &);
};
template<typename Type, bool is_total>
struct on_definedness {
};
template<typename Type>
struct on_definedness<Type, false> {

  // public static functions
  static void add_intersection(Type &, const Type &, const segment_type &);
};
template<typename Type>
struct on_definedness<Type, true> {

  // public static functions
  static void add_intersection(Type &, const Type &, const segment_type &);
};
template<typename Type, bool is_total_invertible>
struct on_invertible {
};
template<typename Type>
struct on_invertible<Type, false> {
  // types
  typedef Type::segment_type             segment_type;
  typedef Type::inverse_codomain_combine inverse_codomain_combine;

  // public static functions
  static void subtract(Type &, const segment_type &);
};
template<typename Type>
struct on_invertible<Type, true> {
  // types
  typedef Type::segment_type             segment_type;
  typedef Type::inverse_codomain_combine inverse_codomain_combine;

  // public static functions
  static void subtract(Type &, const segment_type &);
};
template<typename Type, bool is_total, bool absorbs_identities>
struct on_total_absorbable {
};
template<typename Type, bool absorbs_identities>
struct on_total_absorbable<Type, false, absorbs_identities> {
  // types
```

```
    typedef Type::segment_type                 segment_type;
    typedef Type::codomain_type                codomain_type;
    typedef Type::interval_type                interval_type;
    typedef Type::value_type                   value_type;
    typedef Type::const_iterator               const_iterator;
    typedef Type::set_type                     set_type;
    typedef Type::inverse_codomain_intersect inverse_codomain_intersect;

    // public static functions
    static void flip(Type &, const segment_type &);
  };
  template<typename Type>
  struct on_total_absorbable<Type, true, false> {
    // types
    typedef Type::segment_type  segment_type;
    typedef Type::codomain_type codomain_type;

    // public static functions
    static void flip(Type &, const segment_type &);
  };
  template<typename Type>
  struct on_total_absorbable<Type, true, true> {

    // public static functions
    static void flip(Type &, const typename Type::segment_type &);
  };

  // construct/copy/destruct
  interval_base_map();
  interval_base_map(const interval_base_map &);
  interval_base_map(interval_base_map &&);
  interval_base_map& operator=(const interval_base_map &);
  interval_base_map& operator=(interval_base_map &&);

  // public member functions
  typedef ICL_INTERVAL_TYPE(Interval, DomainT, Compare);
  typedef ICL_COMPARE_DOMAIN(Compare, DomainT);
  typedef ICL_COMPARE_DOMAIN(Compare, segment_type);
  typedef ICL_COMBINE_CODOMAIN(Combine, CodomainT);
   BOOST_STATIC_CONSTANT(bool,
                         is_total_invertible = (Traits::is_total &&has_inverse< codo↵
main_type >::value));
   BOOST_STATIC_CONSTANT(int, fineness = 0);
  void swap(interval_base_map &);
  void clear();
  bool empty() const;
  size_type size() const;
  std::size_t iterative_size() const;
  const_iterator find(const domain_type &) const;
  const_iterator find(const interval_type &) const;
  codomain_type operator()(const domain_type &) const;
  SubType & add(const element_type &);
  SubType & add(const segment_type &);
  iterator add(iterator, const segment_type &);
  SubType & subtract(const element_type &);
  SubType & subtract(const segment_type &);
  SubType & insert(const element_type &);
  SubType & insert(const segment_type &);
  iterator insert(iterator, const segment_type &);
  SubType & set(const element_type &);
  SubType & set(const segment_type &);
  SubType & erase(const element_type &);
  SubType & erase(const segment_type &);
```

```
  SubType & erase(const domain_type &);
  SubType & erase(const interval_type &);
  void erase(iterator);
  void erase(iterator, iterator);
  void add_intersection(SubType &, const segment_type &) const;
  SubType & flip(const element_type &);
  SubType & flip(const segment_type &);
  iterator lower_bound(const key_type &);
  iterator upper_bound(const key_type &);
  const_iterator lower_bound(const key_type &) const;
  const_iterator upper_bound(const key_type &) const;
  std::pair< iterator, iterator > equal_range(const key_type &);
  std::pair< const_iterator, const_iterator >
  equal_range(const key_type &) const;
  iterator begin();
  iterator end();
  const_iterator begin() const;
  const_iterator end() const;
  reverse_iterator rbegin();
  reverse_iterator rend();
  const_reverse_iterator rbegin() const;
  const_reverse_iterator rend() const;
  template<typename Combiner>
    interval_base_map< SubType, DomainT, CodomainT, Traits, Compare, Combine, Section, Inter↵
val, Alloc >::iterator
    _add(const segment_type &);
  template<typename Combiner>
    interval_base_map< SubType, DomainT, CodomainT, Traits, Compare, Combine, Section, Inter↵
val, Alloc >::iterator
    _add(iterator, const segment_type &);

  // private member functions
  template<typename Combiner> iterator _add(const segment_type &);
  template<typename Combiner> iterator _add(iterator, const segment_type &);
  template<typename Combiner> void _subtract(const segment_type &);
  iterator _insert(const segment_type &);
  iterator _insert(iterator, const segment_type &);
  template<typename Combiner>
    void add_segment(const interval_type &, const CodomainT &, iterator &);
  template<typename Combiner>
    void add_main(interval_type &, const CodomainT &, iterator &,
                  const iterator &);
  template<typename Combiner>
    void add_rear(const interval_type &, const CodomainT &, iterator &);
  void add_front(const interval_type &, iterator &);
  void subtract_front(const interval_type &, iterator &);
  template<typename Combiner>
    void subtract_main(const CodomainT &, iterator &, const iterator &);
  template<typename Combiner>
    void subtract_rear(interval_type &, const CodomainT &, iterator &);
  void insert_main(const interval_type &, const CodomainT &, iterator &,
                   const iterator &);
  void erase_rest(interval_type &, const CodomainT &, iterator &,
                  const iterator &);
  template<typename FragmentT>
    void total_add_intersection(SubType &, const FragmentT &) const;
  void partial_add_intersection(SubType &, const segment_type &) const;
  void partial_add_intersection(SubType &, const element_type &) const;

  // protected member functions
  template<typename Combiner>
    iterator gap_insert(iterator, const interval_type &,
                        const codomain_type &);
```

```
  template<typename Combiner>
    std::pair< iterator, bool >
    add_at(const iterator &, const interval_type &, const codomain_type &);
  std::pair< iterator, bool >
  insert_at(const iterator &, const interval_type &, const codomain_type &);
  sub_type * that();
  const sub_type * that() const;
};
```

Description

`interval_base_map` public construct/copy/destruct

1. ```
   interval_base_map();
   ```

   Default constructor for the empty object

2. ```
   interval_base_map(const interval_base_map & src);
   ```

   Copy constructor

3. ```
   interval_base_map(interval_base_map && src);
   ```

   Move constructor

4. ```
   interval_base_map& operator=(const interval_base_map & src);
   ```

   Copy assignment operator

5. ```
   interval_base_map& operator=(interval_base_map && src);
   ```

   Move assignment operator

`interval_base_map` public member functions

1. ```
   typedef ICL_INTERVAL_TYPE(Interval, DomainT, Compare);
   ```

   The interval type of the map.

2. ```
   typedef ICL_COMPARE_DOMAIN(Compare, DomainT);
   ```

   Comparison functor for domain values.

3. ```
   typedef ICL_COMPARE_DOMAIN(Compare, segment_type);
   ```

4. ```
   typedef ICL_COMBINE_CODOMAIN(Combine, CodomainT);
   ```

   Combine functor for codomain value aggregation.
   ```

5.
```
BOOST_STATIC_CONSTANT(bool,
                      is_total_invertible = (Traits::is_total &&has_inverse< codomain_type
>::value));
```

6.
```
BOOST_STATIC_CONSTANT(int, fineness = 0);
```

7.
```
void swap(interval_base_map & object);
```

swap the content of containers

8.
```
void clear();
```

clear the map

9.
```
bool empty() const;
```

is the map empty?

10.
```
size_type size() const;
```

An interval map's size is it's cardinality

11.
```
std::size_t iterative_size() const;
```

Size of the iteration over this container

12.
```
const_iterator find(const domain_type & key_value) const;
```

Find the interval value pair, that contains `key`

13.
```
const_iterator find(const interval_type & key_interval) const;
```

Find the first interval value pair, that collides with interval `key_interval`

14.
```
codomain_type operator()(const domain_type & key_value) const;
```

Total select function.

15.
```
SubType & add(const element_type & key_value_pair);
```

Addition of a key value pair to the map

16.
```
SubType & add(const segment_type & interval_value_pair);
```

Addition of an interval value pair to the map.

17.
```
iterator add(iterator prior_, const segment_type & interval_value_pair);
```

Addition of an interval value pair `interval_value_pair` to the map. Iterator `prior_` is a hint to the position `interval_value_pair` can be inserted after.

18.
```
SubType & subtract(const element_type & key_value_pair);
```

Subtraction of a key value pair from the map

19.
```
SubType & subtract(const segment_type & interval_value_pair);
```

Subtraction of an interval value pair from the map.

20.
```
SubType & insert(const element_type & key_value_pair);
```

Insertion of a `key_value_pair` into the map.

21.
```
SubType & insert(const segment_type & interval_value_pair);
```

Insertion of an `interval_value_pair` into the map.

22.
```
iterator insert(iterator prior, const segment_type & interval_value_pair);
```

Insertion of an `interval_value_pair` into the map. Iterator `prior_`. serves as a hint to insert after the element `prior` point to.

23.
```
SubType & set(const element_type & key_value_pair);
```

With `key_value_pair = (k,v)` set value `v` for key `k`

24.
```
SubType & set(const segment_type & interval_value_pair);
```

With `interval_value_pair = (I,v)` set value `v` for all keys in interval `I` in the map.

25.
```
SubType & erase(const element_type & key_value_pair);
```

Erase a `key_value_pair` from the map.

26.
```
SubType & erase(const segment_type & interval_value_pair);
```

Erase an `interval_value_pair` from the map.

27.
```
SubType & erase(const domain_type & key);
```

Erase a key value pair for `key`.

28.
```
SubType & erase(const interval_type & inter_val);
```

Erase all value pairs within the range of the interval `inter_val` from the map.

29.
```
void erase(iterator position);
```

Erase all value pairs within the range of the interval that iterator `position` points to.

30.
```
void erase(iterator first, iterator past);
```

Erase all value pairs for a range of iterators `[first,past)`.

31.
```
void add_intersection(SubType & section,
                      const segment_type & interval_value_pair) const;
```

The intersection of `interval_value_pair` and `*this` map is added to `section`.

32.
```
SubType & flip(const element_type & key_value_pair);
```

If `*this` map contains `key_value_pair` it is erased, otherwise it is added.

33.
```
SubType & flip(const segment_type & interval_value_pair);
```

If `*this` map contains `interval_value_pair` it is erased, otherwise it is added.

34.
```
iterator lower_bound(const key_type & interval);
```

35.
```
iterator upper_bound(const key_type & interval);
```

36.
```
const_iterator lower_bound(const key_type & interval) const;
```

37.
```
const_iterator upper_bound(const key_type & interval) const;
```

38.
```
std::pair< iterator, iterator > equal_range(const key_type & interval);
```

39.
```
std::pair< const_iterator, const_iterator >
equal_range(const key_type & interval) const;
```

40.
```
iterator begin();
```

41.
```
iterator end();
```

42.
```
const_iterator begin() const;
```

43.
```
const_iterator end() const;
```

44.
```
reverse_iterator rbegin();
```

45.
```
reverse_iterator rend();
```

46.
```
const_reverse_iterator rbegin() const;
```

47.
```
const_reverse_iterator rend() const;
```

48.
```
template<typename Combiner>
  interval_base_map< SubType, DomainT, CodomainT, Traits, Compare, Combine, Section, Interval,
 Alloc >::iterator
  _add(const segment_type & addend);
```

49.
```
template<typename Combiner>
  interval_base_map< SubType, DomainT, CodomainT, Traits, Compare, Combine, Section, Interval,
 Alloc >::iterator
  _add(iterator prior_, const segment_type & addend);
```

`interval_base_map` private member functions

1.
```
template<typename Combiner>
  iterator _add(const segment_type & interval_value_pair);
```

2.
```
template<typename Combiner>
  iterator _add(iterator prior_, const segment_type & interval_value_pair);
```

3.
```
template<typename Combiner>
  void _subtract(const segment_type & interval_value_pair);
```

4.
```
iterator _insert(const segment_type & interval_value_pair);
```

5.
```
iterator _insert(iterator prior_, const segment_type & interval_value_pair);
```

6.
```
template<typename Combiner>
  void add_segment(const interval_type & inter_val, const CodomainT & co_val,
                   iterator & it_);
```

7.
```
template<typename Combiner>
  void add_main(interval_type & inter_val, const CodomainT & co_val,
                iterator & it_, const iterator & last_);
```

8.
```
template<typename Combiner> 
  void add_rear(const interval_type & inter_val, const CodomainT & co_val, 
                iterator & it_);
```

9.
```
void add_front(const interval_type & inter_val, iterator & first_);
```

10.
```
void subtract_front(const interval_type & inter_val, iterator & first_);
```

11.
```
template<typename Combiner> 
  void subtract_main(const CodomainT & co_val, iterator & it_, 
                     const iterator & last_);
```

12.
```
template<typename Combiner> 
  void subtract_rear(interval_type & inter_val, const CodomainT & co_val, 
                     iterator & it_);
```

13.
```
void insert_main(const interval_type &, const CodomainT &, iterator &, 
                 const iterator &);
```

14.
```
void erase_rest(interval_type &, const CodomainT &, iterator &, 
                const iterator &);
```

15.
```
template<typename FragmentT> 
  void total_add_intersection(SubType & section, const FragmentT & fragment) const;
```

16.
```
void partial_add_intersection(SubType & section, const segment_type & operand) const;
```

17.
```
void partial_add_intersection(SubType & section, const element_type & operand) const;
```

`interval_base_map` protected member functions

1.
```
template<typename Combiner> 
  iterator gap_insert(iterator prior_, const interval_type & inter_val, 
                      const codomain_type & co_val);
```

2.
```
template<typename Combiner> 
  std::pair< iterator, bool > 
  add_at(const iterator & prior_, const interval_type & inter_val, 
         const codomain_type & co_val);
```

3.
```
std::pair< iterator, bool >
insert_at(const iterator & prior_, const interval_type & inter_val,
          const codomain_type & co_val);
```

4.
```
sub_type * that();
```

5.
```
const sub_type * that() const;
```

Struct template on_codomain_model

boost::icl::interval_base_map::on_codomain_model

Synopsis

```
// In header: <boost/icl/interval_base_map.hpp>


template<typename Type, bool has_set_semantics>
struct on_codomain_model {
};
```

Struct template on_codomain_model<Type, false>

boost::icl::interval_base_map::on_codomain_model<Type, false>

Synopsis

```
// In header: <boost/icl/interval_base_map.hpp>


template<typename Type>
struct on_codomain_model<Type, false> {
  // types
  typedef Type::interval_type    interval_type;
  typedef Type::codomain_type    codomain_type;
  typedef Type::segment_type     segment_type;
  typedef Type::codomain_combine codomain_combine;

  // public static functions
  static void add(Type &, interval_type &, const codomain_type &,
                  const codomain_type &);
};
```

Description

`on_codomain_model` public static functions

1.
```
static void add(Type & intersection, interval_type & common_interval,
                const codomain_type &, const codomain_type &);
```

Struct template on_codomain_model<Type, true>

boost::icl::interval_base_map::on_codomain_model<Type, true>

Synopsis

```
// In header: <boost/icl/interval_base_map.hpp>


template<typename Type>
struct on_codomain_model<Type, true> {
  // types
  typedef Type::interval_type              interval_type;
  typedef Type::codomain_type              codomain_type;
  typedef Type::segment_type               segment_type;
  typedef Type::codomain_combine           codomain_combine;
  typedef Type::inverse_codomain_intersect inverse_codomain_intersect;

  // public static functions
  static void add(Type &, interval_type &, const codomain_type &,
                  const codomain_type &);
};
```

Description

`on_codomain_model` public static functions

1.
```
static void add(Type & intersection, interval_type & common_interval,
                const codomain_type & flip_value,
                const codomain_type & co_value);
```

Struct template on_definedness

boost::icl::interval_base_map::on_definedness

Synopsis

```
// In header: <boost/icl/interval_base_map.hpp>


template<typename Type, bool is_total>
struct on_definedness {
};
```

Struct template on_definedness<Type, false>

boost::icl::interval_base_map::on_definedness<Type, false>

Synopsis

```
// In header: <boost/icl/interval_base_map.hpp>


template<typename Type>
struct on_definedness<Type, false> {

  // public static functions
  static void add_intersection(Type &, const Type &, const segment_type &);
};
```

Description

`on_definedness` public static functions

1. ```
   static void add_intersection(Type & section, const Type & object,
                                const segment_type & operand);
   ```

Struct template on_definedness<Type, true>

boost::icl::interval_base_map::on_definedness<Type, true>

Synopsis

```
// In header: <boost/icl/interval_base_map.hpp>


template<typename Type>
struct on_definedness<Type, true> {

  // public static functions
  static void add_intersection(Type &, const Type &, const segment_type &);
};
```

Description

`on_definedness` public static functions

1. ```
   static void add_intersection(Type & section, const Type & object,
                                const segment_type & operand);
   ```

Struct template on_invertible

boost::icl::interval_base_map::on_invertible

Synopsis

```
// In header: <boost/icl/interval_base_map.hpp>


template<typename Type, bool is_total_invertible>
struct on_invertible {
};
```

Struct template on_invertible<Type, false>

boost::icl::interval_base_map::on_invertible<Type, false>

Synopsis

```
// In header: <boost/icl/interval_base_map.hpp>


template<typename Type>
struct on_invertible<Type, false> {
  // types
  typedef Type::segment_type              segment_type;
  typedef Type::inverse_codomain_combine inverse_codomain_combine;

  // public static functions
  static void subtract(Type &, const segment_type &);
};
```

Description

`on_invertible` public static functions

1.
```
static void subtract(Type & object, const segment_type & operand);
```

Struct template on_invertible<Type, true>

boost::icl::interval_base_map::on_invertible<Type, true>

Synopsis

```
// In header: <boost/icl/interval_base_map.hpp>


template<typename Type>
struct on_invertible<Type, true> {
  // types
  typedef Type::segment_type              segment_type;
  typedef Type::inverse_codomain_combine inverse_codomain_combine;

  // public static functions
  static void subtract(Type &, const segment_type &);
};
```

Description

`on_invertible` public static functions

1.
```
static void subtract(Type & object, const segment_type & operand);
```

Struct template on_total_absorbable

boost::icl::interval_base_map::on_total_absorbable

Synopsis

```
// In header: <boost/icl/interval_base_map.hpp>


template<typename Type, bool is_total, bool absorbs_identities>
struct on_total_absorbable {
};
```

Struct template on_total_absorbable<Type, false, absorbs_identities>

boost::icl::interval_base_map::on_total_absorbable<Type, false, absorbs_identities>

Synopsis

```
// In header: <boost/icl/interval_base_map.hpp>


template<typename Type, bool absorbs_identities>
struct on_total_absorbable<Type, false, absorbs_identities> {
  // types
  typedef Type::segment_type                segment_type;
  typedef Type::codomain_type               codomain_type;
  typedef Type::interval_type               interval_type;
  typedef Type::value_type                  value_type;
  typedef Type::const_iterator              const_iterator;
  typedef Type::set_type                    set_type;
  typedef Type::inverse_codomain_intersect inverse_codomain_intersect;

  // public static functions
  static void flip(Type &, const segment_type &);
};
```

Description

`on_total_absorbable` public static functions

1.
```
static void flip(Type & object, const segment_type & interval_value_pair);
```

Struct template on_total_absorbable<Type, true, false>

boost::icl::interval_base_map::on_total_absorbable<Type, true, false>

Synopsis

```
// In header: <boost/icl/interval_base_map.hpp>


template<typename Type> 
struct on_total_absorbable<Type, true, false> {
  // types
  typedef Type::segment_type  segment_type; 
  typedef Type::codomain_type codomain_type;

  // public static functions
  static void flip(Type &, const segment_type &);
};
```

Description

`on_total_absorbable` public static functions

1. ```
   static void flip(Type & object, const segment_type & operand);
   ```

Struct template on_total_absorbable<Type, true, true>

boost::icl::interval_base_map::on_total_absorbable<Type, true, true>

Synopsis

```
// In header: <boost/icl/interval_base_map.hpp>


template<typename Type> 
struct on_total_absorbable<Type, true, true> {

  // public static functions
  static void flip(Type &, const typename Type::segment_type &);
};
```

Description

`on_total_absorbable` public static functions

1. ```
   static void flip(Type & object, const typename Type::segment_type &);
   ```

Struct template is_map<icl::interval_base_map< SubType, DomainT, CodomainT, Traits, Compare, Combine, Section, Interval, Alloc >>

boost::icl::is_map<icl::interval_base_map< SubType, DomainT, CodomainT, Traits, Compare, Combine, Section, Interval, Alloc >>

Synopsis

```
// In header: <boost/icl/interval_base_map.hpp>

template<typename SubType, typename DomainT, typename CodomainT, 
         typename Traits, ICL_COMPARE Compare, ICL_COMBINE Combine, 
         ICL_SECTION Section, ICL_INTERVAL(ICL_COMPARE) Interval, 
         ICL_ALLOC Alloc> 
struct is_map<icl::interval_base_map< SubType, DomainT, CodomainT, Traits, Compare, Combine, Sec↵
tion, Interval, Alloc >> {
  // types
  typedef is_map< icl::interval_base_map< SubType, DomainT, CodomainT, Traits, Compare, Com↵
bine, Section, Interval, Alloc > > type;

  // public member functions
   BOOST_STATIC_CONSTANT(bool, value = true);
};
```

Description

`is_map` public member functions

1.
```
BOOST_STATIC_CONSTANT(bool, value = true);
```

Struct template has_inverse<icl::interval_base_map< SubType, DomainT, CodomainT, Traits, Compare, Combine, Section, Interval, Alloc >>

boost::icl::has_inverse<icl::interval_base_map< SubType, DomainT, CodomainT, Traits, Compare, Combine, Section, Interval, Alloc >>

Synopsis

```
// In header: <boost/icl/interval_base_map.hpp>

template<typename SubType, typename DomainT, typename CodomainT, 
         typename Traits, ICL_COMPARE Compare, ICL_COMBINE Combine, 
         ICL_SECTION Section, ICL_INTERVAL(ICL_COMPARE) Interval, 
         ICL_ALLOC Alloc> 
struct has_inverse<icl::interval_base_map< SubType, DomainT, CodomainT, Traits, Compare, Com↵
bine, Section, Interval, Alloc >> {
  // types
  typedef has_inverse< icl::interval_base_map< SubType, DomainT, CodomainT, Traits, Compare, Com↵
bine, Section, Interval, Alloc > > type;

  // public member functions
   BOOST_STATIC_CONSTANT(bool, value = (has_inverse< CodomainT >::value));
};
```

Description

`has_inverse` public member functions

1.
```
BOOST_STATIC_CONSTANT(bool, value = (has_inverse< CodomainT >::value));
```

Struct template is_interval_container<icl::interval_base_map< SubType, DomainT, CodomainT, Traits, Compare, Combine, Section, Interval, Alloc >>

boost::icl::is_interval_container<icl::interval_base_map< SubType, DomainT, CodomainT, Traits, Compare, Combine, Section, Interval, Alloc >>

Synopsis

```
// In header: <boost/icl/interval_base_map.hpp>

template<typename SubType, typename DomainT, typename CodomainT,
         typename Traits, ICL_COMPARE Compare, ICL_COMBINE Combine,
         ICL_SECTION Section, ICL_INTERVAL(ICL_COMPARE) Interval,
         ICL_ALLOC Alloc>
struct is_interval_container<icl::interval_base_map< SubType, DomainT, CodomainT, Traits, Com↵
pare, Combine, Section, Interval, Alloc >> {
  // types
  typedef is_interval_container< icl::interval_base_map< SubType, DomainT, CodomainT, Traits, Com↵
pare, Combine, Section, Interval, Alloc > > type;

  // public member functions
   BOOST_STATIC_CONSTANT(bool, value = true);
};
```

Description

`is_interval_container` public member functions

1.
```
BOOST_STATIC_CONSTANT(bool, value = true);
```

Struct template absorbs_identities<icl::interval_base_map< SubType, DomainT, CodomainT, Traits, Compare, Combine, Section, Interval, Alloc >>

boost::icl::absorbs_identities<icl::interval_base_map< SubType, DomainT, CodomainT, Traits, Compare, Combine, Section, Interval, Alloc >>

Synopsis

```
// In header: <boost/icl/interval_base_map.hpp>

template<typename SubType, typename DomainT, typename CodomainT,
         typename Traits, ICL_COMPARE Compare, ICL_COMBINE Combine,
         ICL_SECTION Section, ICL_INTERVAL(ICL_COMPARE) Interval,
         ICL_ALLOC Alloc>
struct absorbs_identities<icl::interval_base_map< SubType, DomainT, CodomainT, Traits, Compare, Com↵
bine, Section, Interval, Alloc >> {
  // types
  typedef absorbs_identities< icl::interval_base_map< SubType, DomainT, CodomainT, Traits, Com↵
pare, Combine, Section, Interval, Alloc > > type;

  // public member functions
   BOOST_STATIC_CONSTANT(bool, value = (Traits::absorbs_identities));
};
```

Description

`absorbs_identities` public member functions

1. ```
   BOOST_STATIC_CONSTANT(bool, value = (Traits::absorbs_identities));
   ```

Struct template is_total<icl::interval_base_map< SubType, DomainT, CodomainT, Traits, Compare, Combine, Section, Interval, Alloc >>

boost::icl::is_total<icl::interval_base_map< SubType, DomainT, CodomainT, Traits, Compare, Combine, Section, Interval, Alloc >>

Synopsis

```
// In header: <boost/icl/interval_base_map.hpp>

template<typename SubType, typename DomainT, typename CodomainT, 
         typename Traits, ICL_COMPARE Compare, ICL_COMBINE Combine, 
         ICL_SECTION Section, ICL_INTERVAL(ICL_COMPARE) Interval, 
         ICL_ALLOC Alloc> 
struct is_total<icl::interval_base_map< SubType, DomainT, CodomainT, Traits, Compare, Combine, Sec↵
tion, Interval, Alloc >> {
  // types
  typedef is_total< icl::interval_base_map< SubType, DomainT, CodomainT, Traits, Compare, Com↵
bine, Section, Interval, Alloc > > type;

  // public member functions
   BOOST_STATIC_CONSTANT(bool, value = (Traits::is_total));
};
```

Description

`is_total` public member functions

1. ```
   BOOST_STATIC_CONSTANT(bool, value = (Traits::is_total));
   ```

Header <boost/icl/interval_base_set.hpp>

```
namespace boost {
  namespace icl {
    template<typename SubType, typename DomainT, 
             ICL_COMPARE Compare = ICL_COMPARE_INSTANCE(ICL_COMPARE_DEFAULT, DomainT), 
           ICL_INTERVAL(ICL_COMPARE) Interval = ICL_INTERVAL_INSTANCE(ICL_INTERVAL_DEFAULT, Do↵
mainT, Compare), 
             ICL_ALLOC Alloc = std::allocator> 
      class interval_base_set;

    template<typename SubType, typename DomainT, ICL_COMPARE Compare, 
             ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc> 
      struct is_set<icl::interval_base_set< SubType, DomainT, Compare, Interval, Alloc >>;
    template<typename SubType, typename DomainT, ICL_COMPARE Compare, 
             ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc> 
     struct is_interval_container<icl::interval_base_set< SubType, DomainT, Compare, Interval, Al↵
loc >>;
  }
}
```

Class template interval_base_set

boost::icl::interval_base_set — Implements a set as a set of intervals (base class)

Synopsis

```
// In header: <boost/icl/interval_base_set.hpp>

template<typename SubType, typename DomainT, 
         ICL_COMPARE Compare = ICL_COMPARE_INSTANCE(ICL_COMPARE_DEFAULT, DomainT), 
         ICL_INTERVAL(ICL_COMPARE) Interval = ICL_INTERVAL_INSTANCE(ICL_INTERVAL_DEFAULT, Do↵
mainT, Compare), 
         ICL_ALLOC Alloc = std::allocator> 
class interval_base_set {
public:
  // types
  typedef interval_base_set< SubType, DomainT, Compare, Interval, Alloc >   type;           ↵

  typedef SubType                                                          sub_type;         ↵
            // The designated derived or sub_type of this base class.
  typedef type                                                             overloadable_type; ↵
            // Auxilliary type for overloadresolution.
  typedef DomainT                                                          domain_type;      ↵
            // The domain type of the set.
  typedef DomainT                                                          codomain_type;    ↵
            // The codomaintype is the same as domain_type.
  typedef DomainT                                                          element_type;     ↵
            // The element type of the set.
  typedef interval_type                                                    segment_type;     ↵
            // The segment type of the set.
  typedef difference_type_of< domain_type >::type                          difference_type;  ↵
            // The difference type of an interval which is sometimes different form the ↵
data_type.
  typedef size_type_of< domain_type >::type                                size_type;        ↵
            // The size type of an interval which is mostly std::size_t.
  typedef exclusive_less_than< interval_type >                             interval_compare; ↵
            // Comparison functor for intervals.
  typedef exclusive_less_than< interval_type >                             key_compare;      ↵
            // Comparison functor for keys.
  typedef ICL_IMPL_SPACE::set< DomainT, domain_compare, Alloc< DomainT > >  atomized_type;    ↵
            // The atomized type representing the corresponding container of elements.
  typedef Alloc< interval_type >                                           allocator_type;   ↵
            // The allocator type of the set.
  typedef Alloc< DomainT >                                         domain_allocator_type; ↵
          // allocator type of the corresponding element set
  typedef ICL_IMPL_SPACE::set< interval_type, key_compare, allocator_type > ImplSetT;         ↵
            // Container type for the implementation.
  typedef ImplSetT::key_type                                               key_type;         ↵
            // key type of the implementing container
  typedef ImplSetT::key_type                                               data_type;        ↵
            // data type of the implementing container
  typedef ImplSetT::value_type                                             value_type;       ↵
            // value type of the implementing container
  typedef ImplSetT::pointer                                                pointer;          ↵
            // pointer type
  typedef ImplSetT::const_pointer                                          const_pointer;    ↵
            // const pointer type
  typedef ImplSetT::reference                                              reference;        ↵
            // reference type
  typedef ImplSetT::const_reference                                        const_reference;  ↵
            // const reference type
  typedef ImplSetT::iterator                                               iterator;         ↵
            // iterator for iteration over intervals
  typedef ImplSetT::const_iterator                                         const_iterator;   ↵
            // const_iterator for iteration over intervals
  typedef ImplSetT::reverse_iterator                                       reverse_iterator; ↵
```

```
              // iterator for reverse iteration over intervals
  typedef ImplSetT::const_reverse_iterator                                    const_reverse_iter↵
ator;          // const_iterator for iteration over intervals
  typedef boost::icl::element_iterator< iterator >                           element_iterator;  ↵
              // element iterator: Depreciated, see documentation.
  typedef boost::icl::element_iterator< const_iterator >                      element_const_iter↵
ator;          // element const iterator: Depreciated, see documentation.
  typedef boost::icl::element_iterator< reverse_iterator >                  element_reverse_iter↵
ator;        // element reverse iterator: Depreciated, see documentation.
  typedef boost::icl::element_iterator< const_reverse_iterator >             element_const_re↵
verse_iterator;  // element const reverse iterator: Depreciated, see documentation.

  // construct/copy/destruct
  interval_base_set();
  interval_base_set(const interval_base_set &);
  interval_base_set(interval_base_set &&);
  interval_base_set& operator=(const interval_base_set &);
  interval_base_set& operator=(interval_base_set &&);

  // public member functions
  typedef ICL_INTERVAL_TYPE(Interval, DomainT, Compare);
  typedef ICL_COMPARE_DOMAIN(Compare, DomainT);
  typedef ICL_COMPARE_DOMAIN(Compare, segment_type);
   BOOST_STATIC_CONSTANT(int, fineness = 0);
  void swap(interval_base_set &);
  void clear();
  bool empty() const;
  size_type size() const;
  std::size_t iterative_size() const;
  const_iterator find(const element_type &) const;
  const_iterator find(const interval_type &) const;
  SubType & add(const element_type &);
  SubType & add(const segment_type &);
  iterator add(iterator, const segment_type &);
  SubType & subtract(const element_type &);
  SubType & subtract(const segment_type &);
  SubType & insert(const element_type &);
  SubType & insert(const segment_type &);
  iterator insert(iterator, const segment_type &);
  SubType & erase(const element_type &);
  SubType & erase(const segment_type &);
  void erase(iterator);
  void erase(iterator, iterator);
  SubType & flip(const element_type &);
  SubType & flip(const segment_type &);
  iterator begin();
  iterator end();
  const_iterator begin() const;
  const_iterator end() const;
  reverse_iterator rbegin();
  reverse_iterator rend();
  const_reverse_iterator rbegin() const;
  const_reverse_iterator rend() const;
  iterator lower_bound(const value_type &);
  iterator upper_bound(const value_type &);
  const_iterator lower_bound(const value_type &) const;
  const_iterator upper_bound(const value_type &) const;
  std::pair< iterator, iterator > equal_range(const key_type &);
  std::pair< const_iterator, const_iterator >
  equal_range(const key_type &) const;

  // private member functions
  iterator _add(const segment_type &);
```

```
  iterator _add(iterator, const segment_type &);

  // protected member functions
  void add_front(const interval_type &, iterator &);
  void add_main(interval_type &, iterator &, const iterator &);
  void add_segment(const interval_type &, iterator &);
  void add_rear(const interval_type &, iterator &);
  sub_type * that();
  const sub_type * that() const;
};
```

Description

`interval_base_set` public construct/copy/destruct

1. ```
   interval_base_set();
   ```

   Default constructor for the empty object

2. ```
   interval_base_set(const interval_base_set & src);
   ```

   Copy constructor

3. ```
   interval_base_set(interval_base_set && src);
   ```

   Move constructor

4. ```
   interval_base_set& operator=(const interval_base_set & src);
   ```

   Assignment operator

5. ```
   interval_base_set& operator=(interval_base_set && src);
   ```

   Move assignment operator

`interval_base_set` public member functions

1. ```
   typedef ICL_INTERVAL_TYPE(Interval, DomainT, Compare);
   ```

   The interval type of the set.

2. ```
   typedef ICL_COMPARE_DOMAIN(Compare, DomainT);
   ```

   Comparison functor for domain values.

3. ```
   typedef ICL_COMPARE_DOMAIN(Compare, segment_type);
   ```

4. ```
    BOOST_STATIC_CONSTANT(int, fineness = 0);
   ```

5. ```
   void swap(interval_base_set & operand);
   ```

swap the content of containers

6.
```
void clear();
```

sets the container empty

7.
```
bool empty() const;
```

is the container empty?

8.
```
size_type size() const;
```

An interval set's size is it's cardinality

9.
```
std::size_t iterative_size() const;
```

Size of the iteration over this container

10.
```
const_iterator find(const element_type & key_value) const;
```

Find the interval, that contains element `key_value`

11.
```
const_iterator find(const interval_type & key_interval) const;
```

Find the first interval, that collides with interval `key_interval`

12.
```
SubType & add(const element_type & key);
```

Add a single element `key` to the set

13.
```
SubType & add(const segment_type & inter_val);
```

Add an interval of elements `inter_val` to the set

14.
```
iterator add(iterator prior_, const segment_type & inter_val);
```

Add an interval of elements `inter_val` to the set. Iterator `prior_` is a hint to the position `inter_val` can be inserted after.

15.
```
SubType & subtract(const element_type & key);
```

Subtract a single element `key` from the set

16.
```
SubType & subtract(const segment_type & inter_val);
```

Subtract an interval of elements `inter_val` from the set

17.
```
SubType & insert(const element_type & key);
```

Insert an element `key` into the set

18.
```
SubType & insert(const segment_type & inter_val);
```

Insert an interval of elements `inter_val` to the set

19.
```
iterator insert(iterator prior_, const segment_type & inter_val);
```

Insert an interval of elements `inter_val` to the set. Iterator `prior_` is a hint to the position `inter_val` can be inserted after.

20.
```
SubType & erase(const element_type & key);
```

Erase an element `key` from the set

21.
```
SubType & erase(const segment_type & inter_val);
```

Erase an interval of elements `inter_val` from the set

22.
```
void erase(iterator position);
```

Erase the interval that iterator `position` points to.

23.
```
void erase(iterator first, iterator past);
```

Erase all intervals in the range `[first,past)` of iterators.

24.
```
SubType & flip(const element_type & key);
```

If `*this` set contains `key` it is erased, otherwise it is added.

25.
```
SubType & flip(const segment_type & inter_val);
```

If `*this` set contains `inter_val` it is erased, otherwise it is added.

26.
```
iterator begin();
```

27.
```
iterator end();
```

28.
```
const_iterator begin() const;
```

29.
```
const_iterator end() const;
```

30.
```
reverse_iterator rbegin();
```

31.
```
reverse_iterator rend();
```

32.
```
const_reverse_iterator rbegin() const;
```

33.
```
const_reverse_iterator rend() const;
```

34.
```
iterator lower_bound(const value_type & interval);
```

35.
```
iterator upper_bound(const value_type & interval);
```

36.
```
const_iterator lower_bound(const value_type & interval) const;
```

37.
```
const_iterator upper_bound(const value_type & interval) const;
```

38.
```
std::pair< iterator, iterator > equal_range(const key_type & interval);
```

39.
```
std::pair< const_iterator, const_iterator >
equal_range(const key_type & interval) const;
```

`interval_base_set` private member functions

1.
```
iterator _add(const segment_type & addend);
```

2.
```
iterator _add(iterator prior, const segment_type & addend);
```

`interval_base_set` protected member functions

1.
```
void add_front(const interval_type & inter_val, iterator & first_);
```

2.
```
void add_main(interval_type & inter_val, iterator & it_,
              const iterator & last_);
```

3.
```
void add_segment(const interval_type & inter_val, iterator & it_);
```

4.
```
void add_rear(const interval_type & inter_val, iterator & it_);
```

5.
```
sub_type * that();
```

6.
```
const sub_type * that() const;
```

Struct template is_set<icl::interval_base_set< SubType, DomainT, Compare, Interval, Alloc >>

boost::icl::is_set<icl::interval_base_set< SubType, DomainT, Compare, Interval, Alloc >>

Synopsis

```
// In header: <boost/icl/interval_base_set.hpp>

template<typename SubType, typename DomainT, ICL_COMPARE Compare,
         ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc>
struct is_set<icl::interval_base_set< SubType, DomainT, Compare, Interval, Alloc >> {
  // types
  typedef is_set< icl::interval_base_set< SubType, DomainT, Compare, Interval, Alloc > > type;

  // public member functions
   BOOST_STATIC_CONSTANT(bool, value = true);
};
```

Description

`is_set` public member functions

1.
```
BOOST_STATIC_CONSTANT(bool, value = true);
```

Struct template is_interval_container<icl::interval_base_set< SubType, DomainT, Compare, Interval, Alloc >>

boost::icl::is_interval_container<icl::interval_base_set< SubType, DomainT, Compare, Interval, Alloc >>

Synopsis

```
// In header: <boost/icl/interval_base_set.hpp>

template<typename SubType, typename DomainT, ICL_COMPARE Compare,
         ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc>
struct is_interval_container<icl::interval_base_set< SubType, DomainT, Compare, Interval, Al↵
loc >> {
  // types
  typedef is_interval_container< icl::interval_base_set< SubType, DomainT, Compare, Interval, Al↵
loc > > type;

  // public member functions
   BOOST_STATIC_CONSTANT(bool, value = true);
};
```

Description

`is_interval_container` public member functions

1.
```
BOOST_STATIC_CONSTANT(bool, value = true);
```

Header <boost/icl/interval_bounds.hpp>

```
namespace boost {
  namespace icl {
    class interval_bounds;

    typedef unsigned char bound_type;
  }
}
```

Class interval_bounds

boost::icl::interval_bounds

Synopsis

```
// In header: <boost/icl/interval_bounds.hpp>


class interval_bounds {
public:
  // construct/copy/destruct
  interval_bounds();
  explicit interval_bounds(bound_type);

  // public member functions
   BOOST_STATIC_CONSTANT(bound_type, static_open = 0);
   BOOST_STATIC_CONSTANT(bound_type, static_left_open = 1);
   BOOST_STATIC_CONSTANT(bound_type, static_right_open = 2);
   BOOST_STATIC_CONSTANT(bound_type, static_closed = 3);
   BOOST_STATIC_CONSTANT(bound_type, dynamic = 4);
   BOOST_STATIC_CONSTANT(bound_type, undefined = 5);
   BOOST_STATIC_CONSTANT(bound_type, _open = 0);
   BOOST_STATIC_CONSTANT(bound_type, _left_open = 1);
   BOOST_STATIC_CONSTANT(bound_type, _right_open = 2);
   BOOST_STATIC_CONSTANT(bound_type, _closed = 3);
   BOOST_STATIC_CONSTANT(bound_type, _right = 1);
   BOOST_STATIC_CONSTANT(bound_type, _left = 2);
   BOOST_STATIC_CONSTANT(bound_type, _all = 3);
  interval_bounds all() const;
  interval_bounds left() const;
  interval_bounds right() const;
  interval_bounds reverse_left() const;
  interval_bounds reverse_right() const;
  bound_type bits() const;

  // public static functions
  static interval_bounds open();
  static interval_bounds left_open();
  static interval_bounds right_open();
  static interval_bounds closed();

  // public data members
  bound_type _bits;
};
```

Description

`interval_bounds` public construct/copy/destruct

1.
```
interval_bounds();
```

2.
```
explicit interval_bounds(bound_type bounds);
```

`interval_bounds` public member functions

1.
```
BOOST_STATIC_CONSTANT(bound_type, static_open = 0);
```

2.
```
BOOST_STATIC_CONSTANT(bound_type, static_left_open = 1);
```

3.
```
BOOST_STATIC_CONSTANT(bound_type, static_right_open = 2);
```

4.
```
BOOST_STATIC_CONSTANT(bound_type, static_closed = 3);
```

5.
```
BOOST_STATIC_CONSTANT(bound_type, dynamic = 4);
```

6.
```
BOOST_STATIC_CONSTANT(bound_type, undefined = 5);
```

7.
```
BOOST_STATIC_CONSTANT(bound_type, _open = 0);
```

8.
```
BOOST_STATIC_CONSTANT(bound_type, _left_open = 1);
```

9.
```
BOOST_STATIC_CONSTANT(bound_type, _right_open = 2);
```

10.
```
BOOST_STATIC_CONSTANT(bound_type, _closed = 3);
```

11.
```
BOOST_STATIC_CONSTANT(bound_type, _right = 1);
```

12.
```
BOOST_STATIC_CONSTANT(bound_type, _left = 2);
```

13.
```
BOOST_STATIC_CONSTANT(bound_type, _all = 3);
```

14.
```
interval_bounds all() const;
```

15.
```
interval_bounds left() const;
```

16.
```
interval_bounds right() const;
```

17.
```
interval_bounds reverse_left() const;
```

18.
```
interval_bounds reverse_right() const;
```

19.
```
bound_type bits() const;
```

`interval_bounds` public static functions

1.
```
static interval_bounds open();
```

2.
```
static interval_bounds left_open();
```

3.
```
static interval_bounds right_open();
```

4.
```
static interval_bounds closed();
```

Header <boost/icl/interval_combining_style.hpp>

```
namespace boost {
  namespace icl {
    namespace interval_combine {
       BOOST_STATIC_CONSTANT(int, unknown = 0);
       BOOST_STATIC_CONSTANT(int, joining = 1);
       BOOST_STATIC_CONSTANT(int, separating = 2);
       BOOST_STATIC_CONSTANT(int, splitting = 3);
       BOOST_STATIC_CONSTANT(int, elemental = 4);
    }
  }
}
```

Header <boost/icl/interval_map.hpp>

```
namespace boost {
  namespace icl {
    template<typename DomainT, typename CodomainT, typename Traits,
             ICL_COMPARE Compare, ICL_COMBINE Combine, ICL_SECTION Section,
             ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc>
      class split_interval_map;
    template<typename DomainT, typename CodomainT,
             typename Traits = icl::partial_absorber,
             ICL_COMPARE Compare = ICL_COMPARE_INSTANCE(ICL_COMPARE_DEFAULT, DomainT),
             ICL_COMBINE Combine = ICL_COMBINE_INSTANCE(icl::inplace_plus, CodomainT),
             ICL_SECTION Section = ICL_SECTION_INSTANCE(icl::inter_section, CodomainT),
           ICL_INTERVAL(ICL_COMPARE) Interval = ICL_INTERVAL_INSTANCE(ICL_INTERVAL_DEFAULT, Do↲
mainT, Compare),
             ICL_ALLOC Alloc = std::allocator>
      class interval_map;

    template<typename DomainT, typename CodomainT, typename Traits,
             ICL_COMPARE Compare, ICL_COMBINE Combine, ICL_SECTION Section,
             ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc>
     struct is_map<icl::interval_map< DomainT, CodomainT, Traits, Compare, Combine, Section, In↲
terval, Alloc >>;
    template<typename DomainT, typename CodomainT, typename Traits,
             ICL_COMPARE Compare, ICL_COMBINE Combine, ICL_SECTION Section,
             ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc>
      struct has_inverse<icl::interval_map< DomainT, CodomainT, Traits, Compare, Combine, Sec↲
tion, Interval, Alloc >>;
    template<typename DomainT, typename CodomainT, typename Traits,
             ICL_COMPARE Compare, ICL_COMBINE Combine, ICL_SECTION Section,
             ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc>
      struct is_interval_container<icl::interval_map< DomainT, CodomainT, Traits, Compare, Com↲
bine, Section, Interval, Alloc >>;
    template<typename DomainT, typename CodomainT, typename Traits,
             ICL_COMPARE Compare, ICL_COMBINE Combine, ICL_SECTION Section,
             ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc>
      struct absorbs_identities<icl::interval_map< DomainT, CodomainT, Traits, Compare, Com↲
bine, Section, Interval, Alloc >>;
    template<typename DomainT, typename CodomainT, typename Traits,
             ICL_COMPARE Compare, ICL_COMBINE Combine, ICL_SECTION Section,
             ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc>
     struct is_total<icl::interval_map< DomainT, CodomainT, Traits, Compare, Combine, Section, In↲
terval, Alloc >>;
    template<typename DomainT, typename CodomainT, typename Traits,
             ICL_COMPARE Compare, ICL_COMBINE Combine, ICL_SECTION Section,
             ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc>
     struct type_to_string<icl::interval_map< DomainT, CodomainT, Traits, Compare, Combine, Sec↲
tion, Interval, Alloc >>;
  }
}
```

Class template split_interval_map

boost::icl::split_interval_map — implements a map as a map of intervals - on insertion overlapping intervals are split and associated values are combined.

Synopsis

```
// In header: <boost/icl/interval_map.hpp>

template<typename DomainT, typename CodomainT, typename Traits,
         ICL_COMPARE Compare, ICL_COMBINE Combine, ICL_SECTION Section,
         ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc>
class split_interval_map : public boost::icl::interval_base_map< split_interval_map< DomainT, ↵
CodomainT, Traits, Compare, Combine, Section, Interval, Alloc >, DomainT, CodomainT, Traits, Com↵
pare, Combine, Section, Interval, Alloc >
{
public:
  // construct/copy/destruct
  split_interval_map();
  split_interval_map(const split_interval_map &);
  explicit split_interval_map(const domain_mapping_type &);
  explicit split_interval_map(const value_type &);
  split_interval_map(split_interval_map &&);
  split_interval_map& operator=(const split_interval_map &);
  template<typename SubType>
    split_interval_map&
   operator=(const interval_base_map< SubType, DomainT, CodomainT, Traits, Compare, Combine, Sec↵
tion, Interval, Alloc > &);
  split_interval_map& operator=(split_interval_map &&);

  // public member functions
  typedef ICL_INTERVAL_TYPE(Interval, DomainT, Compare);
  template<typename SubType>
   void assign(const interval_base_map< SubType, DomainT, CodomainT, Traits, Compare, Combine, Sec↵
tion, Interval, Alloc > &);

  // private member functions
  iterator handle_inserted(iterator) const;
  void handle_inserted(iterator, iterator) const;
  template<typename Combiner> void handle_left_combined(iterator);
  template<typename Combiner> void handle_combined(iterator);
  template<typename Combiner>
    void handle_preceeded_combined(iterator, iterator &);
  template<typename Combiner>
    void handle_succeeded_combined(iterator, iterator);
  void handle_reinserted(iterator);
  template<typename Combiner>
    void gap_insert_at(iterator &, iterator, const interval_type &,
                       const codomain_type &);
};
```

Description

`split_interval_map` public construct/copy/destruct

1.
```
split_interval_map();
```

   Default constructor for the empty object.

2.
```
split_interval_map(const split_interval_map & src);
```

   Copy constructor.

3.
```
explicit split_interval_map(const domain_mapping_type & base_pair);
```

4.
```
explicit split_interval_map(const value_type & value_pair);
```

5.
```
split_interval_map(split_interval_map && src);
```

Move constructor.

6.
```
split_interval_map& operator=(const split_interval_map & src);
```

Assignment operator.

7.
```
template<typename SubType>
  split_interval_map&
  operator=(const interval_base_map< SubType, DomainT, CodomainT, Traits, Compare, Combine,
Section, Interval, Alloc > & src);
```

Assignment operator for base type.

8.
```
split_interval_map& operator=(split_interval_map && src);
```

Move assignment operator.

`split_interval_map` public member functions

1.
```
typedef ICL_INTERVAL_TYPE(Interval, DomainT, Compare);
```

2.
```
template<typename SubType>
  void assign(const interval_base_map< SubType, DomainT, CodomainT, Traits, Compare, Combine,
 Section, Interval, Alloc > & src);
```

Assignment from a base `interval_map`.

`split_interval_map` private member functions

1.
```
iterator handle_inserted(iterator it_) const;
```

2.
```
void handle_inserted(iterator, iterator) const;
```

3.
```
template<typename Combiner> void handle_left_combined(iterator it_);
```

4.
```
template<typename Combiner> void handle_combined(iterator it_);
```

5.
```
template<typename Combiner>
  void handle_preceeded_combined(iterator prior_, iterator & it_);
```

6.
```
template<typename Combiner>
  void handle_succeeded_combined(iterator it_, iterator);
```

7.
```
void handle_reinserted(iterator);
```

8.
```
template<typename Combiner>
  void gap_insert_at(iterator & it_, iterator prior_,
                     const interval_type & end_gap,
                     const codomain_type & co_val);
```

Class template interval_map

boost::icl::interval_map — implements a map as a map of intervals - on insertion overlapping intervals are split and associated values are combined.

Synopsis

```
// In header: <boost/icl/interval_map.hpp>

template<typename DomainT, typename CodomainT,
         typename Traits = icl::partial_absorber,
         ICL_COMPARE Compare = ICL_COMPARE_INSTANCE(ICL_COMPARE_DEFAULT, DomainT),
         ICL_COMBINE Combine = ICL_COMBINE_INSTANCE(icl::inplace_plus, CodomainT),
         ICL_SECTION Section = ICL_SECTION_INSTANCE(icl::inter_section, CodomainT),
         ICL_INTERVAL(ICL_COMPARE) Interval = ICL_INTERVAL_INSTANCE(ICL_INTERVAL_DEFAULT, Do↵
mainT, Compare),
         ICL_ALLOC Alloc = std::allocator>
class interval_map : public boost::icl::interval_base_map< interval_map< DomainT, CodomainT, ↵
Traits, Compare, Combine, Section, Interval, Alloc >, DomainT, CodomainT, Traits, Compare, Com↵
bine, Section, Interval, Alloc >
{
public:
  // types
  typedef Traits                                                                         ↵
           traits;
  typedef interval_map< DomainT, CodomainT, Traits, Compare, Combine, Section, Interval, Alloc > ↵
         type;
  typedef split_interval_map< DomainT, CodomainT, Traits, Compare, Combine, Section, Interval, Al↵
loc >      split_type;
  typedef type                                                                           ↵
           overloadable_type;
  typedef type                                                                           ↵
           joint_type;
  typedef interval_base_map< type, DomainT, CodomainT, Traits, Compare, Combine, Section, Inter↵
val, Alloc > base_type;
  typedef base_type::iterator                                                            ↵
           iterator;
  typedef base_type::value_type                                                          ↵
           value_type;
  typedef base_type::element_type                                                        ↵
           element_type;
  typedef base_type::segment_type                                                        ↵
           segment_type;
  typedef base_type::domain_type                                                         ↵
           domain_type;
  typedef base_type::codomain_type                                                       ↵
           codomain_type;
  typedef base_type::domain_mapping_type                                                 ↵
           domain_mapping_type;
  typedef base_type::interval_mapping_type                                               ↵
           interval_mapping_type;
  typedef base_type::ImplMapT                                                            ↵
           ImplMapT;
  typedef base_type::size_type                                                           ↵
           size_type;
  typedef base_type::codomain_combine                                                    ↵
           codomain_combine;
  typedef interval_set< DomainT, Compare, Interval, Alloc >                              ↵
           interval_set_type;
  typedef interval_set_type                                                              ↵
           set_type;
  typedef set_type                                                                       ↵
           key_object_type;

  enum @0 { fineness = = 1 };

  // construct/copy/destruct
```

```
  interval_map();
  interval_map(const interval_map &);
  template<typename SubType> 
    explicit interval_map(const interval_base_map< SubType, DomainT, CodomainT, Traits, Com↵
pare, Combine, Section, Interval, Alloc > &);
  explicit interval_map(const domain_mapping_type &);
  explicit interval_map(const value_type &);
  interval_map(interval_map &&);
  interval_map& operator=(const interval_map &);
  template<typename SubType> 
    interval_map& 
   operator=(const interval_base_map< SubType, DomainT, CodomainT, Traits, Compare, Combine, Sec↵
tion, Interval, Alloc > &);
  interval_map& operator=(interval_map &&);

  // public member functions
  typedef ICL_INTERVAL_TYPE(Interval, DomainT, Compare);
  template<typename SubType> 
   void assign(const interval_base_map< SubType, DomainT, CodomainT, Traits, Compare, Combine, Sec↵
tion, Interval, Alloc > &);

  // private member functions
  iterator handle_inserted(iterator);
  void handle_inserted(iterator, iterator);
  template<typename Combiner> void handle_left_combined(iterator);
  template<typename Combiner> void handle_combined(iterator);
  template<typename Combiner> 
    void handle_preceeded_combined(iterator, iterator &);
  template<typename Combiner> 
    void handle_succeeded_combined(iterator, iterator);
  void handle_reinserted(iterator);
  template<typename Combiner> 
    void gap_insert_at(iterator &, iterator, const interval_type &, 
                       const codomain_type &);
};
```

Description

`interval_map` public construct/copy/destruct

1.
```
interval_map();
```

Default constructor for the empty object.

2.
```
interval_map(const interval_map & src);
```

Copy constructor.

3.
```
template<typename SubType> 
  explicit interval_map(const interval_base_map< SubType, DomainT, CodomainT, Traits, Compare, 
 Combine, Section, Interval, Alloc > & src);
```

Copy constructor for base_type.

4.
```
explicit interval_map(const domain_mapping_type & base_pair);
```

5.
```
explicit interval_map(const value_type & value_pair);
```

6.
```
interval_map(interval_map && src);
```

   Move constructor.

7.
```
interval_map& operator=(const interval_map & src);
```

   Assignment operator.

8.
```
template<typename SubType>
  interval_map&
  operator=(const interval_base_map< SubType, DomainT, CodomainT, Traits, Compare, Combine,
Section, Interval, Alloc > & src);
```

   Assignment operator for base type.

9.
```
interval_map& operator=(interval_map && src);
```

   Move assignment operator.

`interval_map` public member functions

1.
```
typedef ICL_INTERVAL_TYPE(Interval, DomainT, Compare);
```

2.
```
template<typename SubType>
  void assign(const interval_base_map< SubType, DomainT, CodomainT, Traits, Compare, Combine,
 Section, Interval, Alloc > & src);
```

   Assignment from a base `interval_map`.

`interval_map` private member functions

1.
```
iterator handle_inserted(iterator it_);
```

2.
```
void handle_inserted(iterator prior_, iterator it_);
```

3.
```
template<typename Combiner> void handle_left_combined(iterator it_);
```

4.
```
template<typename Combiner> void handle_combined(iterator it_);
```

5.
```
template<typename Combiner>
  void handle_preceeded_combined(iterator prior_, iterator & it_);
```

6.
```
template<typename Combiner>
  void handle_succeeded_combined(iterator it_, iterator next_);
```

7.
```
void handle_reinserted(iterator insertion_);
```

8.
```
template<typename Combiner>
  void gap_insert_at(iterator & it_, iterator prior_,
                     const interval_type & end_gap,
                     const codomain_type & co_val);
```

Struct template is_map<icl::interval_map< DomainT, CodomainT, Traits, Compare, Combine, Section, Interval, Alloc >>

boost::icl::is_map<icl::interval_map< DomainT, CodomainT, Traits, Compare, Combine, Section, Interval, Alloc >>

Synopsis

```
// In header: <boost/icl/interval_map.hpp>

template<typename DomainT, typename CodomainT, typename Traits,
         ICL_COMPARE Compare, ICL_COMBINE Combine, ICL_SECTION Section,
         ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc>
struct is_map<icl::interval_map< DomainT, CodomainT, Traits, Compare, Combine, Section, Inter↵
val, Alloc >> {
  // types
  typedef is_map< icl::interval_map< DomainT, CodomainT, Traits, Compare, Combine, Section, In↵
terval, Alloc > > type;

  // public member functions
   BOOST_STATIC_CONSTANT(bool, value = true);
};
```

Description

`is_map` public member functions

1.
```
BOOST_STATIC_CONSTANT(bool, value = true);
```

Struct template has_inverse<icl::interval_map< DomainT, CodomainT, Traits, Compare, Combine, Section, Interval, Alloc >>

boost::icl::has_inverse<icl::interval_map< DomainT, CodomainT, Traits, Compare, Combine, Section, Interval, Alloc >>

Synopsis

```
// In header: <boost/icl/interval_map.hpp>

template<typename DomainT, typename CodomainT, typename Traits, 
         ICL_COMPARE Compare, ICL_COMBINE Combine, ICL_SECTION Section, 
         ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc> 
struct has_inverse<icl::interval_map< DomainT, CodomainT, Traits, Compare, Combine, Section, In↵
terval, Alloc >> {
  // types
  typedef has_inverse< icl::interval_map< DomainT, CodomainT, Traits, Compare, Combine, Sec↵
tion, Interval, Alloc > > type;

  // public member functions
   BOOST_STATIC_CONSTANT(bool, value = (has_inverse< CodomainT >::value));
};
```

Description

`has_inverse` public member functions

1.
```
BOOST_STATIC_CONSTANT(bool, value = (has_inverse< CodomainT >::value));
```

Struct template is_interval_container<icl::interval_map< DomainT, CodomainT, Traits, Compare, Combine, Section, Interval, Alloc >>

boost::icl::is_interval_container<icl::interval_map< DomainT, CodomainT, Traits, Compare, Combine, Section, Interval, Alloc >>

Synopsis

```
// In header: <boost/icl/interval_map.hpp>

template<typename DomainT, typename CodomainT, typename Traits, 
         ICL_COMPARE Compare, ICL_COMBINE Combine, ICL_SECTION Section, 
         ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc> 
struct is_interval_container<icl::interval_map< DomainT, CodomainT, Traits, Compare, Combine, Sec↵
tion, Interval, Alloc >> {
  // types
  typedef is_interval_container< icl::interval_map< DomainT, CodomainT, Traits, Compare, Com↵
bine, Section, Interval, Alloc > > type;

  // public member functions
   BOOST_STATIC_CONSTANT(bool, value = true);
};
```

Description

`is_interval_container` public member functions

1.
```
BOOST_STATIC_CONSTANT(bool, value = true);
```

Struct template absorbs_identities<icl::interval_map< DomainT, CodomainT, Traits, Compare, Combine, Section, Interval, Alloc >>

boost::icl::absorbs_identities<icl::interval_map< DomainT, CodomainT, Traits, Compare, Combine, Section, Interval, Alloc >>

Synopsis

```
// In header: <boost/icl/interval_map.hpp>

template<typename DomainT, typename CodomainT, typename Traits,
         ICL_COMPARE Compare, ICL_COMBINE Combine, ICL_SECTION Section,
         ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc>
struct absorbs_identities<icl::interval_map< DomainT, CodomainT, Traits, Compare, Combine, Sec↵
tion, Interval, Alloc >> {
  // types
  typedef absorbs_identities< icl::interval_map< DomainT, CodomainT, Traits, Compare, Combine, Sec↵
tion, Interval, Alloc > > type;

  // public member functions
   BOOST_STATIC_CONSTANT(bool, value = (Traits::absorbs_identities));
};
```

Description

`absorbs_identities` public member functions

1.
```
BOOST_STATIC_CONSTANT(bool, value = (Traits::absorbs_identities));
```

Struct template is_total<icl::interval_map< DomainT, CodomainT, Traits, Compare, Combine, Section, Interval, Alloc >>

boost::icl::is_total<icl::interval_map< DomainT, CodomainT, Traits, Compare, Combine, Section, Interval, Alloc >>

Synopsis

```
// In header: <boost/icl/interval_map.hpp>

template<typename DomainT, typename CodomainT, typename Traits,
         ICL_COMPARE Compare, ICL_COMBINE Combine, ICL_SECTION Section,
         ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc>
struct is_total<icl::interval_map< DomainT, CodomainT, Traits, Compare, Combine, Section, Inter↵
val, Alloc >> {
  // types
  typedef is_total< icl::interval_map< DomainT, CodomainT, Traits, Compare, Combine, Section, In↵
terval, Alloc > > type;

  // public member functions
   BOOST_STATIC_CONSTANT(bool, value = (Traits::is_total));
};
```

Description

`is_total` public member functions

1.
```
BOOST_STATIC_CONSTANT(bool, value = (Traits::is_total));
```

Struct template type_to_string<icl::interval_map< DomainT, CodomainT, Traits, Compare, Combine, Section, Interval, Alloc >>

boost::icl::type_to_string<icl::interval_map< DomainT, CodomainT, Traits, Compare, Combine, Section, Interval, Alloc >>

Synopsis

```
// In header: <boost/icl/interval_map.hpp>

template<typename DomainT, typename CodomainT, typename Traits, 
         ICL_COMPARE Compare, ICL_COMBINE Combine, ICL_SECTION Section, 
         ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc> 
struct type_to_string<icl::interval_map< DomainT, CodomainT, Traits, Compare, Combine, Section, In↲
terval, Alloc >> {

  // public static functions
  static std::string apply();
};
```

Description

`type_to_string` public static functions

1.
```
static std::string apply();
```

Header <boost/icl/interval_set.hpp>

```
namespace boost {
  namespace icl {
    template<typename DomainT,
             ICL_COMPARE Compare = ICL_COMPARE_INSTANCE(ICL_COMPARE_DEFAULT, DomainT),
           ICL_INTERVAL(ICL_COMPARE) Interval = ICL_INTERVAL_INSTANCE(ICL_INTERVAL_DEFAULT, Do↵
mainT, Compare),
             ICL_ALLOC Alloc = std::allocator>
      class interval_set;

    template<typename DomainT, ICL_COMPARE Compare,
             ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc>
      struct is_set<icl::interval_set< DomainT, Compare, Interval, Alloc >>;
    template<typename DomainT, ICL_COMPARE Compare,
             ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc>
      struct is_interval_container<icl::interval_set< DomainT, Compare, Interval, Alloc >>;
    template<typename DomainT, ICL_COMPARE Compare,
             ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc>
      struct is_interval_joiner<icl::interval_set< DomainT, Compare, Interval, Alloc >>;
    template<typename DomainT, ICL_COMPARE Compare,
             ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc>
      struct type_to_string<icl::interval_set< DomainT, Compare, Interval, Alloc >>;
  }
}
```

Class template interval_set

boost::icl::interval_set — Implements a set as a set of intervals - merging adjoining intervals.

Synopsis

```
// In header: <boost/icl/interval_set.hpp>

template<typename DomainT,
         ICL_COMPARE Compare = ICL_COMPARE_INSTANCE(ICL_COMPARE_DEFAULT, DomainT),
         ICL_INTERVAL(ICL_COMPARE) Interval = ICL_INTERVAL_INSTANCE(ICL_INTERVAL_DEFAULT, Do↵
mainT, Compare),
         ICL_ALLOC Alloc = std::allocator>
class interval_set : public boost::icl::interval_base_set< interval_set< DomainT, Compare, Inter↵
val, Alloc >, DomainT, Compare, Interval, Alloc >
{
public:
  // types
  typedef interval_set< DomainT, Compare, Interval, Alloc >            type;
  typedef interval_base_set< type, DomainT, Compare, Interval, Alloc > base_type;          ↵
 // The base_type of this class.
  typedef type                                                          overloadable_type;
  typedef type                                                          joint_type;
  typedef type                                                          key_object_type;
  typedef DomainT                                                      domain_type;          ↵
 // The domain type of the set.
  typedef DomainT                                                      codomain_type;        ↵
 // The codomaintype is the same as domain_type.
  typedef DomainT                                                      element_type;         ↵
 // The element type of the set.
  typedef interval_type                                                segment_type;         ↵
 // The segment type of the set.
  typedef exclusive_less_than< interval_type >                         interval_compare;     ↵
 // Comparison functor for intervals.
  typedef exclusive_less_than< interval_type >                         key_compare;          ↵
 // Comparison functor for keys.
  typedef Alloc< interval_type >                                       allocator_type;       ↵
 // The allocator type of the set.
  typedef Alloc< DomainT >                                             domain_allocator_type; ↵
 // allocator type of the corresponding element set
  typedef base_type::atomized_type                                     atomized_type;        ↵
 // The corresponding atomized type representing this interval container of elements.
  typedef base_type::ImplSetT                                          ImplSetT;             ↵
 // Container type for the implementation.
  typedef ImplSetT::key_type                                           key_type;             ↵
 // key type of the implementing container
  typedef ImplSetT::value_type                                         data_type;            ↵
 // data type of the implementing container
  typedef ImplSetT::value_type                                         value_type;           ↵
 // value type of the implementing container
  typedef ImplSetT::iterator                                           iterator;             ↵
 // iterator for iteration over intervals
  typedef ImplSetT::const_iterator                                     const_iterator;       ↵
 // const_iterator for iteration over intervals

  enum @1 { fineness = = 1 };

  // construct/copy/destruct
  interval_set();
  interval_set(const interval_set &);
  template<typename SubType>
    explicit interval_set(const interval_base_set< SubType, DomainT, Compare, Interval, Alloc > &);
  explicit interval_set(const domain_type &);
  explicit interval_set(const interval_type &);
  interval_set(interval_set &&);
  interval_set& operator=(const interval_set &);
```

```
  template<typename SubType>
    interval_set&
    operator=(const interval_base_set< SubType, DomainT, Compare, Interval, Alloc > &);
  interval_set& operator=(interval_set &&);

  // public member functions
  typedef ICL_INTERVAL_TYPE(Interval, DomainT, Compare);
  typedef ICL_COMPARE_DOMAIN(Compare, DomainT);
  template<typename SubType>
    void assign(const interval_base_set< SubType, DomainT, Compare, Interval, Alloc > &);

  // private member functions
  iterator handle_inserted(iterator);
  iterator add_over(const interval_type &, iterator);
  iterator add_over(const interval_type &);
};
```

Description

`interval_set` public construct/copy/destruct

1.
```
interval_set();
```

   Default constructor for the empty object.

2.
```
interval_set(const interval_set & src);
```

   Copy constructor.

3.
```
template<typename SubType>
  explicit interval_set(const interval_base_set< SubType, DomainT, Compare, Interval, Alloc
> & src);
```

   Copy constructor for base_type.

4.
```
explicit interval_set(const domain_type & value);
```

   Constructor for a single element.

5.
```
explicit interval_set(const interval_type & itv);
```

   Constructor for a single interval.

6.
```
interval_set(interval_set && src);
```

   Move constructor.

7.
```
interval_set& operator=(const interval_set & src);
```

   Assignment operator.

8.
```
template<typename SubType>
  interval_set&
  operator=(const interval_base_set< SubType, DomainT, Compare, Interval, Alloc > & src);
```

Assignment operator for base type.

9.
```
interval_set& operator=(interval_set && src);
```

Move assignment operator.

`interval_set` public member functions

1.
```
typedef ICL_INTERVAL_TYPE(Interval, DomainT, Compare);
```

The interval type of the set.

2.
```
typedef ICL_COMPARE_DOMAIN(Compare, DomainT);
```

Comparison functor for domain values.

3.
```
template<typename SubType>
  void assign(const interval_base_set< SubType, DomainT, Compare, Interval, Alloc > & src);
```

Assignment from a base `interval_set`.

`interval_set` private member functions

1.
```
iterator handle_inserted(iterator it_);
```

2.
```
iterator add_over(const interval_type & addend, iterator last_);
```

3.
```
iterator add_over(const interval_type & addend);
```

Struct template is_set<icl::interval_set< DomainT, Compare, Interval, Alloc >>

boost::icl::is_set<icl::interval_set< DomainT, Compare, Interval, Alloc >>

Synopsis

```
// In header: <boost/icl/interval_set.hpp>

template<typename DomainT, ICL_COMPARE Compare,
         ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc>
struct is_set<icl::interval_set< DomainT, Compare, Interval, Alloc >> {
  // types
  typedef is_set< icl::interval_set< DomainT, Compare, Interval, Alloc > > type;

  // public member functions
   BOOST_STATIC_CONSTANT(bool, value = true);
};
```

Description

`is_set` public member functions

1.
```
BOOST_STATIC_CONSTANT(bool, value = true);
```

Struct template is_interval_container<icl::interval_set< DomainT, Compare, Interval, Alloc >>

boost::icl::is_interval_container<icl::interval_set< DomainT, Compare, Interval, Alloc >>

Synopsis

```
// In header: <boost/icl/interval_set.hpp>

template<typename DomainT, ICL_COMPARE Compare,
         ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc>
struct is_interval_container<icl::interval_set< DomainT, Compare, Interval, Alloc >> {
  // types
  typedef is_interval_container< icl::interval_set< DomainT, Compare, Interval, Alloc > > type;

  // public member functions
   BOOST_STATIC_CONSTANT(bool, value = true);
};
```

Description

`is_interval_container` public member functions

1.
```
BOOST_STATIC_CONSTANT(bool, value = true);
```

Struct template is_interval_joiner<icl::interval_set< DomainT, Compare, Interval, Alloc >>

boost::icl::is_interval_joiner<icl::interval_set< DomainT, Compare, Interval, Alloc >>

Synopsis

```
// In header: <boost/icl/interval_set.hpp>

template<typename DomainT, ICL_COMPARE Compare,
         ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc>
struct is_interval_joiner<icl::interval_set< DomainT, Compare, Interval, Alloc >> {
  // types
  typedef is_interval_joiner< icl::interval_set< DomainT, Compare, Interval, Alloc > > type;

  // public member functions
   BOOST_STATIC_CONSTANT(bool, value = true);
};
```

Description

`is_interval_joiner` public member functions

1.
```
BOOST_STATIC_CONSTANT(bool, value = true);
```

Struct template type_to_string<icl::interval_set< DomainT, Compare, Interval, Alloc >>

boost::icl::type_to_string<icl::interval_set< DomainT, Compare, Interval, Alloc >>

Synopsis

```
// In header: <boost/icl/interval_set.hpp>

template<typename DomainT, ICL_COMPARE Compare, 
         ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc> 
struct type_to_string<icl::interval_set< DomainT, Compare, Interval, Alloc >> {

  // public static functions
  static std::string apply();
};
```

Description

`type_to_string` public static functions

1.
```
static std::string apply();
```

Header <boost/icl/interval_traits.hpp>

```
namespace boost {
  namespace icl {
    template<typename Type> struct interval_traits;

    template<typename Type> struct domain_type_of<interval_traits< Type >>;
    template<typename Type> struct difference_type_of<interval_traits< Type >>;
    template<typename Type> struct size_type_of<interval_traits< Type >>;
  }
}
```

Struct template interval_traits

boost::icl::interval_traits

Synopsis

```
// In header: <boost/icl/interval_traits.hpp>

template<typename Type>
struct interval_traits {
  // types
  typedef interval_traits                   type;
  typedef domain_type_of< Type >::type domain_type;

  // public static functions
  static Type construct(const domain_type &, const domain_type &);
  static domain_type upper(const Type &);
  static domain_type lower(const Type &);
};
```

Description

`interval_traits` public static functions

1.
```
static Type construct(const domain_type & lo, const domain_type & up);
```

2.
```
static domain_type upper(const Type & inter_val);
```

3.
```
static domain_type lower(const Type & inter_val);
```

Struct template domain_type_of<interval_traits< Type >>

boost::icl::domain_type_of<interval_traits< Type >>

Synopsis

```
// In header: <boost/icl/interval_traits.hpp>

template<typename Type>
struct domain_type_of<interval_traits< Type >> {
  // types
  typedef interval_traits< Type >::domain_type type;
};
```

Struct template difference_type_of<interval_traits< Type >>

boost::icl::difference_type_of<interval_traits< Type >>

Synopsis

```
// In header: <boost/icl/interval_traits.hpp>

template<typename Type>
struct difference_type_of<interval_traits< Type >> {
  // types
  typedef interval_traits< Type >::domain_type    domain_type;
  typedef difference_type_of< domain_type >::type type;
};
```

Struct template size_type_of<interval_traits< Type >>

boost::icl::size_type_of<interval_traits< Type >>

Synopsis

```
// In header: <boost/icl/interval_traits.hpp>

template<typename Type>
struct size_type_of<interval_traits< Type >> {
  // types
  typedef interval_traits< Type >::domain_type domain_type;
  typedef size_type_of< domain_type >::type    type;
};
```

Header <boost/icl/iterator.hpp>

```
namespace boost {
  namespace icl {
    template<typename ContainerT> class add_iterator;
    template<typename ContainerT> class insert_iterator;
    template<typename ContainerT, typename IteratorT>
      add_iterator< ContainerT > adder(ContainerT &, IteratorT);
    template<typename ContainerT, typename IteratorT>
      insert_iterator< ContainerT > inserter(ContainerT &, IteratorT);
  }
}
```

Class template add_iterator

boost::icl::add_iterator — Performes an addition using a container's memberfunction add, when operator= is called.

Synopsis

```
// In header: <boost/icl/iterator.hpp>

template<typename ContainerT>
class add_iterator :
  public std::iterator< std::output_iterator_tag, void, void, void, void >
{
public:
  // types
  typedef ContainerT                  container_type;     // The container's type.
  typedef std::output_iterator_tag iterator_category;

  // construct/copy/destruct
  add_iterator(ContainerT &, typename ContainerT::iterator);
  add_iterator& operator=(typename ContainerT::const_reference);

  // public member functions
  add_iterator & operator*();
  add_iterator & operator++();
  add_iterator & operator++(int);
};
```

Description

`add_iterator` public construct/copy/destruct

1.
```
add_iterator(ContainerT & cont, typename ContainerT::iterator iter);
```

An `add_iterator` is constructed with a container and a position that has to be maintained.

2.
```
add_iterator& operator=(typename ContainerT::const_reference value);
```

This assignment operator adds the `value` before the current position. It maintains it's position by incrementing after addition.

`add_iterator` public member functions

1.
```
add_iterator & operator*();
```

2.
```
add_iterator & operator++();
```

3.
```
add_iterator & operator++(int);
```

Class template insert_iterator

boost::icl::insert_iterator — Performes an insertion using a container's memberfunction add, when operator= is called.

Synopsis

```
// In header: <boost/icl/iterator.hpp>

template<typename ContainerT>
class insert_iterator :
  public std::iterator< std::output_iterator_tag, void, void, void, void >
{
public:
  // types
  typedef ContainerT                 container_type;     // The container's type.
  typedef std::output_iterator_tag iterator_category;

  // construct/copy/destruct
  insert_iterator(ContainerT &, typename ContainerT::iterator);
  insert_iterator& operator=(typename ContainerT::const_reference);

  // public member functions
  insert_iterator & operator*();
  insert_iterator & operator++();
  insert_iterator & operator++(int);
};
```

Description

`insert_iterator` public construct/copy/destruct

1.
```
insert_iterator(ContainerT & cont, typename ContainerT::iterator iter);
```

An `insert_iterator` is constructed with a container and a position that has to be maintained.

2.
```
insert_iterator& operator=(typename ContainerT::const_reference value);
```

This assignment operator adds the `value` before the current position. It maintains it's position by incrementing after addition.

`insert_iterator` public member functions

1.
```
insert_iterator & operator*();
```

2.
```
insert_iterator & operator++();
```

3.
```
insert_iterator & operator++(int);
```

Function template adder

boost::icl::adder

Synopsis

```
// In header: <boost/icl/iterator.hpp>


template<typename ContainerT, typename IteratorT>
  add_iterator< ContainerT > adder(ContainerT & cont, IteratorT iter_);
```

Description

Function adder creates and initializes an `add_iterator`

Function template inserter

boost::icl::inserter

Synopsis

```
// In header: <boost/icl/iterator.hpp>


template<typename ContainerT, typename IteratorT>
  insert_iterator< ContainerT > inserter(ContainerT & cont, IteratorT iter_);
```

Description

Function inserter creates and initializes an `insert_iterator`

Header <boost/icl/left_open_interval.hpp>

```
namespace boost {
  namespace icl {
    template<typename DomainT,
             ICL_COMPARE Compare = ICL_COMPARE_INSTANCE(ICL_COMPARE_DEFAULT, DomainT)>
      class left_open_interval;

    template<typename DomainT, ICL_COMPARE Compare>
      struct interval_traits<icl::left_open_interval< DomainT, Compare >>;
    template<typename DomainT, ICL_COMPARE Compare>
      struct interval_bound_type<left_open_interval< DomainT, Compare >>;
    template<typename DomainT, ICL_COMPARE Compare>
      struct type_to_string<icl::left_open_interval< DomainT, Compare >>;
    template<typename DomainT, ICL_COMPARE Compare>
      struct value_size<icl::left_open_interval< DomainT, Compare >>;
  }
}
```

Class template left_open_interval

boost::icl::left_open_interval

Synopsis

```
// In header: <boost/icl/left_open_interval.hpp>

template<typename DomainT,
         ICL_COMPARE Compare = ICL_COMPARE_INSTANCE(ICL_COMPARE_DEFAULT, DomainT)>
class left_open_interval {
public:
  // types
  typedef left_open_interval< DomainT, Compare > type;
  typedef DomainT                                domain_type;

  // construct/copy/destruct
  left_open_interval();
  explicit left_open_interval(const DomainT &);
  left_open_interval(const DomainT &, const DomainT &);

  // public member functions
  typedef ICL_COMPARE_DOMAIN(Compare, DomainT);
  DomainT lower() const;
  DomainT upper() const;
};
```

Description

`left_open_interval` public construct/copy/destruct

1. ```
   left_open_interval();
   ```

   Default constructor; yields an empty interval `(0,0]`.

2. ```
   explicit left_open_interval(const DomainT & val);
   ```

   Constructor for a left-open singleton interval `(val-1,val]`

3. ```
   left_open_interval(const DomainT & low, const DomainT & up);
   ```

   Interval from `low` to `up` with bounds `bounds`

`left_open_interval` public member functions

1. ```
   typedef ICL_COMPARE_DOMAIN(Compare, DomainT);
   ```

2. ```
   DomainT lower() const;
   ```

3. ```
   DomainT upper() const;
   ```

Struct template interval_traits<icl::left_open_interval< DomainT, Compare >>

boost::icl::interval_traits<icl::left_open_interval< DomainT, Compare >>

Synopsis

```
// In header: <boost/icl/left_open_interval.hpp>

template<typename DomainT, ICL_COMPARE Compare> 
struct interval_traits<icl::left_open_interval< DomainT, Compare >> {
  // types
  typedef DomainT                                        domain_type;  
  typedef icl::left_open_interval< DomainT, Compare > interval_type;

  // public member functions
  typedef ICL_COMPARE_DOMAIN(Compare, DomainT);

  // public static functions
  static interval_type construct(const domain_type &, const domain_type &);
  static domain_type lower(const interval_type &);
  static domain_type upper(const interval_type &);
};
```

Description

`interval_traits` public member functions

1.
```
typedef ICL_COMPARE_DOMAIN(Compare, DomainT);
```

`interval_traits` public static functions

1.
```
static interval_type construct(const domain_type & lo, const domain_type & up);
```

2.
```
static domain_type lower(const interval_type & inter_val);
```

3.
```
static domain_type upper(const interval_type & inter_val);
```

Struct template interval_bound_type<left_open_interval< DomainT, Compare >>

boost::icl::interval_bound_type<left_open_interval< DomainT, Compare >>

Synopsis

```
// In header: <boost/icl/left_open_interval.hpp>

template<typename DomainT, ICL_COMPARE Compare> 
struct interval_bound_type<left_open_interval< DomainT, Compare >> {
  // types
  typedef interval_bound_type type;

  // public member functions
   BOOST_STATIC_CONSTANT(bound_type, 
                         value = interval_bounds::static_left_open);
};
```

Description

`interval_bound_type` public member functions

1.
```
BOOST_STATIC_CONSTANT(bound_type, value = interval_bounds::static_left_open);
```

Struct template type_to_string<icl::left_open_interval< DomainT, Compare >>

boost::icl::type_to_string<icl::left_open_interval< DomainT, Compare >>

Synopsis

```
// In header: <boost/icl/left_open_interval.hpp>

template<typename DomainT, ICL_COMPARE Compare>
struct type_to_string<icl::left_open_interval< DomainT, Compare >> {

  // public static functions
  static std::string apply();
};
```

Description

`type_to_string` public static functions

1.
```
static std::string apply();
```

Struct template value_size<icl::left_open_interval< DomainT, Compare >>

boost::icl::value_size<icl::left_open_interval< DomainT, Compare >>

Synopsis

```
// In header: <boost/icl/left_open_interval.hpp>

template<typename DomainT, ICL_COMPARE Compare>
struct value_size<icl::left_open_interval< DomainT, Compare >> {

  // public static functions
  static std::size_t apply(const icl::left_open_interval< DomainT > &);
};
```

Description

`value_size` public static functions

1.
```
static std::size_t apply(const icl::left_open_interval< DomainT > &);
```

Header <boost/icl/map.hpp>

```
namespace boost {
  namespace icl {
    struct partial_absorber;
    struct partial_enricher;
    struct total_absorber;
    struct total_enricher;

    template<typename DomainT, typename CodomainT,
             typename Traits = icl::partial_absorber,
             ICL_COMPARE Compare = ICL_COMPARE_INSTANCE(ICL_COMPARE_DEFAULT, DomainT),
             ICL_COMBINE Combine = ICL_COMBINE_INSTANCE(icl::inplace_plus, CodomainT),
             ICL_SECTION Section = ICL_SECTION_INSTANCE(icl::inter_section, CodomainT),
             ICL_ALLOC Alloc = std::allocator>
      class map;

    template<typename DomainT, typename CodomainT, typename Traits,
             ICL_COMPARE Compare, ICL_COMBINE Combine, ICL_SECTION Section,
             ICL_ALLOC Alloc>
      struct is_map<icl::map< DomainT, CodomainT, Traits, Compare, Combine, Section, Alloc >>;
    template<typename DomainT, typename CodomainT, typename Traits,
             ICL_COMPARE Compare, ICL_COMBINE Combine, ICL_SECTION Section,
             ICL_ALLOC Alloc>
     struct has_inverse<icl::map< DomainT, CodomainT, Traits, Compare, Combine, Section, Alloc >>;
    template<typename DomainT, typename CodomainT, typename Traits,
             ICL_COMPARE Compare, ICL_COMBINE Combine, ICL_SECTION Section,
             ICL_ALLOC Alloc>
      struct absorbs_identities<icl::map< DomainT, CodomainT, Traits, Compare, Combine, Sec↵
tion, Alloc >>;
    template<typename DomainT, typename CodomainT, typename Traits,
             ICL_COMPARE Compare, ICL_COMBINE Combine, ICL_SECTION Section,
             ICL_ALLOC Alloc>
     struct is_total<icl::map< DomainT, CodomainT, Traits, Compare, Combine, Section, Alloc >>;
    template<typename DomainT, typename CodomainT, typename Traits,
             ICL_COMPARE Compare, ICL_COMBINE Combine, ICL_SECTION Section,
             ICL_ALLOC Alloc>
     struct type_to_string<icl::map< DomainT, CodomainT, Traits, Compare, Combine, Section, Al↵
loc >>;
  }
}
```

Struct partial_absorber

boost::icl::partial_absorber

Synopsis

```
// In header: <boost/icl/map.hpp>


struct partial_absorber {

  enum @2 { absorbs_identities = = true };

  enum @3 { is_total = = false };
};
```

Struct partial_enricher

boost::icl::partial_enricher

Synopsis

```
// In header: <boost/icl/map.hpp>


struct partial_enricher {

  enum @4 { absorbs_identities = = false };

  enum @5 { is_total = = false };
};
```

Struct total_absorber

boost::icl::total_absorber

Synopsis

```
// In header: <boost/icl/map.hpp>


struct total_absorber {

  enum @6 { absorbs_identities = = true };

  enum @7 { is_total = = true };
};
```

Struct total_enricher

boost::icl::total_enricher

Synopsis

```
// In header: <boost/icl/map.hpp>


struct total_enricher {

  enum @8 { absorbs_identities = = false };

  enum @9 { is_total = = true };
};
```

Class template map

boost::icl::map — Addable, subractable and intersectable maps.

Synopsis

```
// In header: <boost/icl/map.hpp>

template<typename DomainT, typename CodomainT,
         typename Traits = icl::partial_absorber,
         ICL_COMPARE Compare = ICL_COMPARE_INSTANCE(ICL_COMPARE_DEFAULT, DomainT),
         ICL_COMBINE Combine = ICL_COMBINE_INSTANCE(icl::inplace_plus, CodomainT),
         ICL_SECTION Section = ICL_SECTION_INSTANCE(icl::inter_section, CodomainT),
         ICL_ALLOC Alloc = std::allocator>
class map : private ICL_IMPL_SPACE::map< DomainT, CodomainT, ICL_COMPARE_DOMAIN(Compare, Do↵
mainT), Alloc< std::pair< const DomainT, CodomainT > > >
{
public:
  // types
  typedef Alloc< typename std::pair< const DomainT, CodomainT > >                  ↵
                            allocator_type;
  typedef icl::map< DomainT, CodomainT, Traits, Compare, Combine, Section, Alloc >    ↵
                            type;
  typedef ICL_IMPL_SPACE::map< DomainT, CodomainT, ICL_COMPARE_DOMAIN(Compare, DomainT), alloc↵
ator_type >                  base_type;
  typedef Traits                                                                      ↵
                            traits;
  typedef DomainT                                                                     ↵
                            domain_type;
  typedef boost::call_traits< DomainT >::param_type                                   ↵
                            domain_param;
  typedef DomainT                                                                     ↵
                            key_type;
  typedef CodomainT                                                                   ↵
                            codomain_type;
  typedef CodomainT                                                                   ↵
                            mapped_type;
  typedef CodomainT                                                                   ↵
                            data_type;
  typedef std::pair< const DomainT, CodomainT >                                       ↵
                            element_type;
  typedef std::pair< const DomainT, CodomainT >                                       ↵
                            value_type;
  typedef domain_compare                                                              ↵
                            key_compare;
  typedef inverse< codomain_combine >::type                                           ↵
                            inverse_codomain_combine;
  typedef mpl::if_< has_set_semantics< codomain_type >, ICL_SECTION_CODOMAIN(Section, Codo↵
mainT), codomain_combine >::type codomain_intersect;
  typedef inverse< codomain_intersect >::type                                         ↵
                            inverse_codomain_intersect;
  typedef base_type::value_compare                                                    ↵
                            value_compare;
  typedef ICL_IMPL_SPACE::set< DomainT, domain_compare, Alloc< DomainT > >            ↵
                            set_type;
  typedef set_type                                                                    ↵
                            key_object_type;
  typedef on_absorbtion< type, codomain_combine, Traits::absorbs_identities >         ↵
                            on_identity_absorbtion;
  typedef base_type::pointer                                                          ↵
                            pointer;
  typedef base_type::const_pointer                                                    ↵
                            const_pointer;
  typedef base_type::reference                                                        ↵
                            reference;
  typedef base_type::const_reference                                                  ↵
```

```
                            const_reference;
typedef base_type::iterator                                                                      ↵
                            iterator;
typedef base_type::const_iterator                                                                ↵
                            const_iterator;
typedef base_type::size_type                                                                     ↵
                            size_type;
typedef base_type::difference_type                                                               ↵
                            difference_type;
typedef base_type::reverse_iterator                                                              ↵
                            reverse_iterator;
typedef base_type::const_reverse_iterator                                                        ↵
                            const_reverse_iterator;

// member classes/structs/unions
template<typename Type, bool has_set_semantics, bool absorbs_identities>
struct on_codomain_model {
};
template<typename Type>
struct on_codomain_model<Type, false, false> {

  // public static functions
  static void subtract(Type &, typename Type::iterator,
                       const typename Type::codomain_type &);
};
template<typename Type>
struct on_codomain_model<Type, false, true> {

  // public static functions
  static void subtract(Type &, typename Type::iterator,
                       const typename Type::codomain_type &);
};
template<typename Type>
struct on_codomain_model<Type, true, false> {
  // types
  typedef Type::inverse_codomain_intersect inverse_codomain_intersect;

  // public static functions
  static void subtract(Type &, typename Type::iterator,
                       const typename Type::codomain_type &);
};
template<typename Type>
struct on_codomain_model<Type, true, true> {
  // types
  typedef Type::inverse_codomain_intersect inverse_codomain_intersect;

  // public static functions
  static void subtract(Type &, typename Type::iterator,
                       const typename Type::codomain_type &);
};
template<typename Type, bool is_total>
struct on_definedness {
};
template<typename Type>
struct on_definedness<Type, false> {

  // public static functions
  static void add_intersection(Type &, const Type &, const element_type &);
};
template<typename Type>
struct on_definedness<Type, true> {

  // public static functions
```

```
  static void add_intersection(Type &, const Type &, const element_type &);
};
template<typename Type, bool is_total_invertible> 
struct on_invertible {
};
template<typename Type> 
struct on_invertible<Type, false> {
  // types
  typedef Type::element_type              element_type;           
  typedef Type::inverse_codomain_combine inverse_codomain_combine;

  // public static functions
  static void subtract(Type &, const element_type &);
};
template<typename Type> 
struct on_invertible<Type, true> {
  // types
  typedef Type::element_type              element_type;           
  typedef Type::inverse_codomain_combine inverse_codomain_combine;

  // public static functions
  static void subtract(Type &, const element_type &);
};
template<typename Type, bool is_total, bool absorbs_identities> 
struct on_total_absorbable {
};
template<typename Type> 
struct on_total_absorbable<Type, false, false> {
  // types
  typedef Type::element_type                element_type;             
  typedef Type::codomain_type               codomain_type;            
  typedef Type::iterator                    iterator;                 
  typedef Type::inverse_codomain_intersect inverse_codomain_intersect;

  // public static functions
  static void flip(Type &, const element_type &);
};
template<typename Type> 
struct on_total_absorbable<Type, false, true> {
  // types
  typedef Type::element_type                element_type;             
  typedef Type::codomain_type               codomain_type;            
  typedef Type::iterator                    iterator;                 
  typedef Type::inverse_codomain_intersect inverse_codomain_intersect;

  // public static functions
  static void flip(Type &, const element_type &);
};
template<typename Type> 
struct on_total_absorbable<Type, true, false> {
  // types
  typedef Type::element_type  element_type; 
  typedef Type::codomain_type codomain_type;

  // public static functions
  static void flip(Type &, const element_type &);
};
template<typename Type> 
struct on_total_absorbable<Type, true, true> {
  // types
  typedef Type::element_type element_type;

  // public static functions
```

```
    static void flip(Type &, const typename Type::element_type &);
  };

  // construct/copy/destruct
  map();
  map(const key_compare &);
  template<typename InputIterator> map(InputIterator, InputIterator);
  template<typename InputIterator>
    map(InputIterator, InputIterator, const key_compare &);
  map(const map &);
  explicit map(const element_type &);
  map(map &&);
  map& operator=(const map &);
  map& operator=(map &&);

  // public member functions
  typedef ICL_COMPARE_DOMAIN(Compare, DomainT);
  typedef ICL_COMBINE_CODOMAIN(Combine, CodomainT);
  typedef ICL_COMPARE_DOMAIN(Compare, element_type);
   BOOST_STATIC_CONSTANT(bool, _total = (Traits::is_total));
   BOOST_STATIC_CONSTANT(bool, _absorbs = (Traits::absorbs_identities));
   BOOST_STATIC_CONSTANT(bool,
                          total_invertible = (mpl::and_< is_total< type >, has_inverse< codo↵
main_type > >::value));
   BOOST_STATIC_CONSTANT(bool,
                          is_total_invertible = (Traits::is_total &&has_inverse< codo↵
main_type >::value));
   BOOST_STATIC_CONSTANT(int, fineness = 4);
  void swap(map &);
  template<typename SubObject> bool contains(const SubObject &) const;
  bool within(const map &) const;
  std::size_t iterative_size() const;
  codomain_type operator()(const domain_type &) const;
  map & add(const value_type &);
  iterator add(iterator, const value_type &);
  map & subtract(const element_type &);
  map & subtract(const domain_type &);
  std::pair< iterator, bool > insert(const value_type &);
  iterator insert(iterator, const value_type &);
  template<typename Iterator> iterator insert(Iterator, Iterator);
  map & set(const element_type &);
  size_type erase(const element_type &);
  void add_intersection(map &, const element_type &) const;
  map & flip(const element_type &);
  template<typename Combiner>
    map< DomainT, CodomainT, Traits, Compare, Combine, Section, Alloc > &
    _add(const element_type &);
  template<typename Combiner>
    map< DomainT, CodomainT, Traits, Compare, Combine, Section, Alloc >::iterator
    _add(iterator, const value_type &);
  template<typename Combiner>
    map< DomainT, CodomainT, Traits, Compare, Combine, Section, Alloc > &
    _subtract(const value_type &);

  // private member functions
  template<typename Combiner> map & _add(const element_type &);
  template<typename Combiner> iterator _add(iterator, const element_type &);
  template<typename Combiner> map & _subtract(const element_type &);
  template<typename FragmentT>
    void total_add_intersection(type &, const FragmentT &) const;
  void partial_add_intersection(type &, const element_type &) const;
};
```

Description

`map` public construct/copy/destruct

1.
```
map();
```

2.
```
map(const key_compare & comp);
```

3.
```
template<typename InputIterator> map(InputIterator first, InputIterator past);
```

4.
```
template<typename InputIterator>
  map(InputIterator first, InputIterator past, const key_compare & comp);
```

5.
```
map(const map & src);
```

6.
```
explicit map(const element_type & key_value_pair);
```

7.
```
map(map && src);
```

8.
```
map& operator=(const map & src);
```

9.
```
map& operator=(map && src);
```

`map` public member functions

1.
```
typedef ICL_COMPARE_DOMAIN(Compare, DomainT);
```

2.
```
typedef ICL_COMBINE_CODOMAIN(Combine, CodomainT);
```

3.
```
typedef ICL_COMPARE_DOMAIN(Compare, element_type);
```

4.
```
 BOOST_STATIC_CONSTANT(bool, _total = (Traits::is_total));
```

5.
```
 BOOST_STATIC_CONSTANT(bool, _absorbs = (Traits::absorbs_identities));
```

6.
```
BOOST_STATIC_CONSTANT(bool,
                      total_invertible = (mpl::and_< is_total< type >, has_inverse< codo↵
main_type > >::value));
```

7.
```
BOOST_STATIC_CONSTANT(bool,
                      is_total_invertible = (Traits::is_total &&has_inverse< codomain_type
>::value));
```

8.
```
BOOST_STATIC_CONSTANT(int, fineness = 4);
```

9.
```
void swap(map & src);
```

10.
```
template<typename SubObject> bool contains(const SubObject & sub) const;
```

11.
```
bool within(const map & super) const;
```

12.
```
std::size_t iterative_size() const;
```

`iterative_size()` yields the number of elements that is visited throu complete iteration. For interval sets `iterative_size()` is different from `size()`.

13.
```
codomain_type operator()(const domain_type & key) const;
```

Total select function.

14.
```
map & add(const value_type & value_pair);
```

`add` inserts `value_pair` into the map if it's key does not exist in the map. If `value_pairs's` key value exists in the map, it's data value is added to the data value already found in the map.

15.
```
iterator add(iterator prior, const value_type & value_pair);
```

`add` add `value_pair` into the map using `prior` as a hint to insert `value_pair` after the position `prior` is pointing to.

16.
```
map & subtract(const element_type & value_pair);
```

If the `value_pair's` key value is in the map, it's data value is subtraced from the data value stored in the map.

17.
```
map & subtract(const domain_type & key);
```

18.
```
std::pair< iterator, bool > insert(const value_type & value_pair);
```

19.
```
iterator insert(iterator prior, const value_type & value_pair);
```

20.
```
template<typename Iterator> iterator insert(Iterator first, Iterator last);
```

21.
```
map & set(const element_type & key_value_pair);
```

With `key_value_pair = (k,v)` set value `v` for key `k`

22.
```
size_type erase(const element_type & key_value_pair);
```

erase `key_value_pair` from the map. Erase only if, the exact value content `val` is stored for the given key.

23.
```
void add_intersection(map & section, const element_type & key_value_pair) const;
```

The intersection of `key_value_pair` and `*this` map is added to `section`.

24.
```
map & flip(const element_type & operand);
```

25.
```
template<typename Combiner> 
  map< DomainT, CodomainT, Traits, Compare, Combine, Section, Alloc > & 
  _add(const element_type & addend);
```

26.
```
template<typename Combiner> 
  map< DomainT, CodomainT, Traits, Compare, Combine, Section, Alloc >::iterator 
  _add(iterator prior_, const value_type & addend);
```

27.
```
template<typename Combiner> 
  map< DomainT, CodomainT, Traits, Compare, Combine, Section, Alloc > & 
  _subtract(const value_type & minuend);
```

`map` private member functions

1.
```
template<typename Combiner> map & _add(const element_type & value_pair);
```

2.
```
template<typename Combiner> 
  iterator _add(iterator prior, const element_type & value_pair);
```

3.
```
template<typename Combiner> map & _subtract(const element_type & value_pair);
```

4.
```
template<typename FragmentT> 
  void total_add_intersection(type & section, const FragmentT & fragment) const;
```

5.
```
void partial_add_intersection(type & section, const element_type & operand) const;
```

Struct template on_codomain_model

boost::icl::map::on_codomain_model

Synopsis

```
// In header: <boost/icl/map.hpp>


template<typename Type, bool has_set_semantics, bool absorbs_identities>
struct on_codomain_model {
};
```

Struct template on_codomain_model<Type, false, false>

boost::icl::map::on_codomain_model<Type, false, false>

Synopsis

```
// In header: <boost/icl/map.hpp>


template<typename Type>
struct on_codomain_model<Type, false, false> {

  // public static functions
  static void subtract(Type &, typename Type::iterator,
                       const typename Type::codomain_type &);
};
```

Description

`on_codomain_model` public static functions

1.
```
static void subtract(Type &, typename Type::iterator it_,
                     const typename Type::codomain_type &);
```

Struct template on_codomain_model<Type, false, true>

boost::icl::map::on_codomain_model<Type, false, true>

Synopsis

```
// In header: <boost/icl/map.hpp>


template<typename Type>
struct on_codomain_model<Type, false, true> {

  // public static functions
  static void subtract(Type &, typename Type::iterator,
                       const typename Type::codomain_type &);
};
```

Description

`on_codomain_model` public static functions

1. ```
   static void subtract(Type & object, typename Type::iterator it_,
                        const typename Type::codomain_type &);
   ```

Struct template on_codomain_model<Type, true, false>

boost::icl::map::on_codomain_model<Type, true, false>

Synopsis

```
// In header: <boost/icl/map.hpp>


template<typename Type>
struct on_codomain_model<Type, true, false> {
  // types
  typedef Type::inverse_codomain_intersect inverse_codomain_intersect;

  // public static functions
  static void subtract(Type &, typename Type::iterator,
                       const typename Type::codomain_type &);
};
```

Description

`on_codomain_model` public static functions

1. ```
   static void subtract(Type &, typename Type::iterator it_,
                        const typename Type::codomain_type & co_value);
   ```

Struct template on_codomain_model<Type, true, true>

boost::icl::map::on_codomain_model<Type, true, true>

Synopsis

```
// In header: <boost/icl/map.hpp>


template<typename Type>
struct on_codomain_model<Type, true, true> {
  // types
  typedef Type::inverse_codomain_intersect inverse_codomain_intersect;

  // public static functions
  static void subtract(Type &, typename Type::iterator,
                       const typename Type::codomain_type &);
};
```

Description

`on_codomain_model` public static functions

1.
```
static void subtract(Type & object, typename Type::iterator it_,
                     const typename Type::codomain_type & co_value);
```

Struct template on_definedness

boost::icl::map::on_definedness

Synopsis

```
// In header: <boost/icl/map.hpp>


template<typename Type, bool is_total>
struct on_definedness {
};
```

Struct template on_definedness<Type, false>

boost::icl::map::on_definedness<Type, false>

Synopsis

```
// In header: <boost/icl/map.hpp>


template<typename Type>
struct on_definedness<Type, false> {

  // public static functions
  static void add_intersection(Type &, const Type &, const element_type &);
};
```

Description

`on_definedness` public static functions

1.
```
static void add_intersection(Type & section, const Type & object, 
                             const element_type & operand);
```

Struct template on_definedness<Type, true>

boost::icl::map::on_definedness<Type, true>

Synopsis

```
// In header: <boost/icl/map.hpp>


template<typename Type> 
struct on_definedness<Type, true> {

  // public static functions
  static void add_intersection(Type &, const Type &, const element_type &);
};
```

Description

`on_definedness` public static functions

1.
```
static void add_intersection(Type & section, const Type & object, 
                             const element_type & operand);
```

Struct template on_invertible

boost::icl::map::on_invertible

Synopsis

```
// In header: <boost/icl/map.hpp>


template<typename Type, bool is_total_invertible> 
struct on_invertible {
};
```

Struct template on_invertible<Type, false>

boost::icl::map::on_invertible<Type, false>

Synopsis

```
// In header: <boost/icl/map.hpp>


template<typename Type> 
struct on_invertible<Type, false> {
  // types
  typedef Type::element_type              element_type;            
  typedef Type::inverse_codomain_combine inverse_codomain_combine;

  // public static functions
  static void subtract(Type &, const element_type &);
};
```

Description

`on_invertible` public static functions

1.
```
static void subtract(Type & object, const element_type & operand);
```

Struct template on_invertible<Type, true>

boost::icl::map::on_invertible<Type, true>

Synopsis

```
// In header: <boost/icl/map.hpp>


template<typename Type> 
struct on_invertible<Type, true> {
  // types
  typedef Type::element_type              element_type;            
  typedef Type::inverse_codomain_combine inverse_codomain_combine;

  // public static functions
  static void subtract(Type &, const element_type &);
};
```

Description

`on_invertible` public static functions

1.
```
static void subtract(Type & object, const element_type & operand);
```

Struct template on_total_absorbable

boost::icl::map::on_total_absorbable

Synopsis

```
// In header: <boost/icl/map.hpp>


template<typename Type, bool is_total, bool absorbs_identities>
struct on_total_absorbable {
};
```

Struct template on_total_absorbable<Type, false, false>

boost::icl::map::on_total_absorbable<Type, false, false>

Synopsis

```
// In header: <boost/icl/map.hpp>


template<typename Type>
struct on_total_absorbable<Type, false, false> {
  // types
  typedef Type::element_type                element_type;
  typedef Type::codomain_type               codomain_type;
  typedef Type::iterator                    iterator;
  typedef Type::inverse_codomain_intersect inverse_codomain_intersect;

  // public static functions
  static void flip(Type &, const element_type &);
};
```

Description

`on_total_absorbable` public static functions

1.
```
static void flip(Type & object, const element_type & operand);
```

Struct template on_total_absorbable<Type, false, true>

boost::icl::map::on_total_absorbable<Type, false, true>

Synopsis

```
// In header: <boost/icl/map.hpp>


template<typename Type>
struct on_total_absorbable<Type, false, true> {
  // types
  typedef Type::element_type                element_type;
  typedef Type::codomain_type               codomain_type;
  typedef Type::iterator                    iterator;
  typedef Type::inverse_codomain_intersect inverse_codomain_intersect;

  // public static functions
  static void flip(Type &, const element_type &);
};
```

Description

`on_total_absorbable` public static functions

1.
```
static void flip(Type & object, const element_type & operand);
```

Struct template on_total_absorbable<Type, true, false>

boost::icl::map::on_total_absorbable<Type, true, false>

Synopsis

```
// In header: <boost/icl/map.hpp>


template<typename Type>
struct on_total_absorbable<Type, true, false> {
  // types
  typedef Type::element_type  element_type;
  typedef Type::codomain_type codomain_type;

  // public static functions
  static void flip(Type &, const element_type &);
};
```

Description

`on_total_absorbable` public static functions

1.
```
static void flip(Type & object, const element_type & operand);
```

Struct template on_total_absorbable<Type, true, true>

boost::icl::map::on_total_absorbable<Type, true, true>

Synopsis

```
// In header: <boost/icl/map.hpp>


template<typename Type>
struct on_total_absorbable<Type, true, true> {
  // types
  typedef Type::element_type element_type;

  // public static functions
  static void flip(Type &, const typename Type::element_type &);
};
```

Description

`on_total_absorbable` public static functions

1.
```
static void flip(Type & object, const typename Type::element_type &);
```

Struct template is_map<icl::map< DomainT, CodomainT, Traits, Compare, Combine, Section, Alloc >>

boost::icl::is_map<icl::map< DomainT, CodomainT, Traits, Compare, Combine, Section, Alloc >>

Synopsis

```
// In header: <boost/icl/map.hpp>

template<typename DomainT, typename CodomainT, typename Traits,
         ICL_COMPARE Compare, ICL_COMBINE Combine, ICL_SECTION Section,
         ICL_ALLOC Alloc>
struct is_map<icl::map< DomainT, CodomainT, Traits, Compare, Combine, Section, Alloc >> {
  // types
  typedef is_map< icl::map< DomainT, CodomainT, Traits, Compare, Combine, Section, Alloc > > type;

  // public member functions
   BOOST_STATIC_CONSTANT(bool, value = true);
};
```

Description

`is_map` public member functions

1.
```
 BOOST_STATIC_CONSTANT(bool, value = true);
```

Struct template has_inverse<icl::map< DomainT, CodomainT, Traits, Compare, Combine, Section, Alloc >>

boost::icl::has_inverse<icl::map< DomainT, CodomainT, Traits, Compare, Combine, Section, Alloc >>

Synopsis

```
// In header: <boost/icl/map.hpp>

template<typename DomainT, typename CodomainT, typename Traits,
         ICL_COMPARE Compare, ICL_COMBINE Combine, ICL_SECTION Section,
         ICL_ALLOC Alloc>
struct has_inverse<icl::map< DomainT, CodomainT, Traits, Compare, Combine, Section, Alloc >> {
  // types
  typedef has_inverse< icl::map< DomainT, CodomainT, Traits, Compare, Combine, Section, Al↵
loc > > type;

  // public member functions
   BOOST_STATIC_CONSTANT(bool, value = (has_inverse< CodomainT >::value));
};
```

Description

has_inverse public member functions

1.
```
BOOST_STATIC_CONSTANT(bool, value = (has_inverse< CodomainT >::value));
```

Struct template absorbs_identities<icl::map< DomainT, CodomainT, Traits, Compare, Combine, Section, Alloc >>

boost::icl::absorbs_identities<icl::map< DomainT, CodomainT, Traits, Compare, Combine, Section, Alloc >>

Synopsis

```
// In header: <boost/icl/map.hpp>

template<typename DomainT, typename CodomainT, typename Traits,
         ICL_COMPARE Compare, ICL_COMBINE Combine, ICL_SECTION Section,
         ICL_ALLOC Alloc>
struct absorbs_identities<icl::map< DomainT, CodomainT, Traits, Compare, Combine, Section, Al↵
loc >> {
  // types
  typedef absorbs_identities type;

  // public member functions
   BOOST_STATIC_CONSTANT(int, value = Traits::absorbs_identities);
};
```

Description

absorbs_identities public member functions

1.
```
BOOST_STATIC_CONSTANT(int, value = Traits::absorbs_identities);
```

Struct template is_total<icl::map< DomainT, CodomainT, Traits, Compare, Combine, Section, Alloc >>

boost::icl::is_total<icl::map< DomainT, CodomainT, Traits, Compare, Combine, Section, Alloc >>

Synopsis

```
// In header: <boost/icl/map.hpp>

template<typename DomainT, typename CodomainT, typename Traits,
         ICL_COMPARE Compare, ICL_COMBINE Combine, ICL_SECTION Section,
         ICL_ALLOC Alloc>
struct is_total<icl::map< DomainT, CodomainT, Traits, Compare, Combine, Section, Alloc >> {
  // types
  typedef is_total type;

  // public member functions
   BOOST_STATIC_CONSTANT(int, value = Traits::is_total);
};
```

Description

`is_total` public member functions

1.
```
BOOST_STATIC_CONSTANT(int, value = Traits::is_total);
```

Struct template type_to_string<icl::map< DomainT, CodomainT, Traits, Compare, Combine, Section, Alloc >>

boost::icl::type_to_string<icl::map< DomainT, CodomainT, Traits, Compare, Combine, Section, Alloc >>

Synopsis

```
// In header: <boost/icl/map.hpp>

template<typename DomainT, typename CodomainT, typename Traits,
         ICL_COMPARE Compare, ICL_COMBINE Combine, ICL_SECTION Section,
         ICL_ALLOC Alloc>
struct type_to_string<icl::map< DomainT, CodomainT, Traits, Compare, Combine, Section, Alloc >> {

  // public static functions
  static std::string apply();
};
```

Description

`type_to_string` public static functions

1.
```
static std::string apply();
```

Header <boost/icl/open_interval.hpp>

```
namespace boost {
  namespace icl {
    template<typename DomainT,
             ICL_COMPARE Compare = ICL_COMPARE_INSTANCE(ICL_COMPARE_DEFAULT, DomainT)>
      class open_interval;

    template<typename DomainT, ICL_COMPARE Compare>
      struct interval_traits<icl::open_interval< DomainT, Compare >>;
    template<typename DomainT, ICL_COMPARE Compare>
      struct interval_bound_type<open_interval< DomainT, Compare >>;
    template<typename DomainT, ICL_COMPARE Compare>
      struct type_to_string<icl::open_interval< DomainT, Compare >>;
    template<typename DomainT, ICL_COMPARE Compare>
      struct value_size<icl::open_interval< DomainT, Compare >>;
  }
}
```

Class template open_interval

boost::icl::open_interval

Synopsis

```
// In header: <boost/icl/open_interval.hpp>

template<typename DomainT,
         ICL_COMPARE Compare = ICL_COMPARE_INSTANCE(ICL_COMPARE_DEFAULT, DomainT)>
class open_interval {
public:
  // types
  typedef open_interval< DomainT, Compare > type;
  typedef DomainT                           domain_type;

  // construct/copy/destruct
  open_interval();
  explicit open_interval(const DomainT &);
  open_interval(const DomainT &, const DomainT &);

  // public member functions
  typedef ICL_COMPARE_DOMAIN(Compare, DomainT);
  DomainT lower() const;
  DomainT upper() const;
};
```

Description

`open_interval` public construct/copy/destruct

1.
```
open_interval();
```

Default constructor; yields an empty interval `(0,0)`.

2.
```
explicit open_interval(const DomainT & val);
```

Constructor for an open singleton interval `(val-1,val+1)`

3.
```
open_interval(const DomainT & low, const DomainT & up);
```

Interval from low to up with bounds bounds

open_interval public member functions

1.
```
typedef ICL_COMPARE_DOMAIN(Compare, DomainT);
```

2.
```
DomainT lower() const;
```

3.
```
DomainT upper() const;
```

Struct template interval_traits<icl::open_interval< DomainT, Compare >>

boost::icl::interval_traits<icl::open_interval< DomainT, Compare >>

Synopsis

```
// In header: <boost/icl/open_interval.hpp>

template<typename DomainT, ICL_COMPARE Compare>
struct interval_traits<icl::open_interval< DomainT, Compare >> {
  // types
  typedef DomainT                                          domain_type;
  typedef icl::open_interval< DomainT, Compare > interval_type;

  // public member functions
  typedef ICL_COMPARE_DOMAIN(Compare, DomainT);

  // public static functions
  static interval_type construct(const domain_type &, const domain_type &);
  static domain_type lower(const interval_type &);
  static domain_type upper(const interval_type &);
};
```

Description

interval_traits public member functions

1.
```
typedef ICL_COMPARE_DOMAIN(Compare, DomainT);
```

interval_traits public static functions

1.
```
static interval_type construct(const domain_type & lo, const domain_type & up);
```

2.
```
static domain_type lower(const interval_type & inter_val);
```

3.
```
static domain_type upper(const interval_type & inter_val);
```

Struct template interval_bound_type<open_interval< DomainT, Compare >>

boost::icl::interval_bound_type<open_interval< DomainT, Compare >>

Synopsis

```
// In header: <boost/icl/open_interval.hpp>

template<typename DomainT, ICL_COMPARE Compare>
struct interval_bound_type<open_interval< DomainT, Compare >> {
  // types
  typedef interval_bound_type type;

  // public member functions
   BOOST_STATIC_CONSTANT(bound_type, value = interval_bounds::static_open);
};
```

Description

`interval_bound_type` public member functions

1.
```
 BOOST_STATIC_CONSTANT(bound_type, value = interval_bounds::static_open);
```

Struct template type_to_string<icl::open_interval< DomainT, Compare >>

boost::icl::type_to_string<icl::open_interval< DomainT, Compare >>

Synopsis

```
// In header: <boost/icl/open_interval.hpp>

template<typename DomainT, ICL_COMPARE Compare>
struct type_to_string<icl::open_interval< DomainT, Compare >> {

  // public static functions
  static std::string apply();
};
```

Description

`type_to_string` public static functions

1.
```
static std::string apply();
```

Struct template value_size<icl::open_interval< DomainT, Compare >>

boost::icl::value_size<icl::open_interval< DomainT, Compare >>

Synopsis

```
// In header: <boost/icl/open_interval.hpp>

template<typename DomainT, ICL_COMPARE Compare>
struct value_size<icl::open_interval< DomainT, Compare >> {

  // public static functions
  static std::size_t apply(const icl::open_interval< DomainT > &);
};
```

Description

`value_size` public static functions

1.
```
static std::size_t apply(const icl::open_interval< DomainT > &);
```

Header <boost/icl/ptime.hpp>

```
namespace boost {
  namespace icl {
    template<> struct is_discrete<boost::posix_time::ptime>;
    template<> struct has_difference<boost::posix_time::ptime>;
    template<> struct difference_type_of<boost::posix_time::ptime>;
    template<> struct size_type_of<boost::posix_time::ptime>;
    template<> struct is_discrete<boost::posix_time::time_duration>;
    template<> struct has_difference<boost::posix_time::time_duration>;
    template<> struct size_type_of<boost::posix_time::time_duration>;
    boost::posix_time::ptime operator++(boost::posix_time::ptime & x);
    boost::posix_time::ptime operator--(boost::posix_time::ptime & x);
    boost::posix_time::time_duration
    operator++(boost::posix_time::time_duration & x);
    boost::posix_time::time_duration
    operator--(boost::posix_time::time_duration & x);
  }
}
```

Struct is_discrete<boost::posix_time::ptime>

boost::icl::is_discrete<boost::posix_time::ptime>

Synopsis

```
// In header: <boost/icl/ptime.hpp>


struct is_discrete<boost::posix_time::ptime> {
  // types
  typedef is_discrete type;

  // public member functions
   BOOST_STATIC_CONSTANT(bool, value = true);
};
```

Description

`is_discrete` public member functions

1.
```
BOOST_STATIC_CONSTANT(bool, value = true);
```

Struct has_difference<boost::posix_time::ptime>

boost::icl::has_difference<boost::posix_time::ptime>

Synopsis

```
// In header: <boost/icl/ptime.hpp>


struct has_difference<boost::posix_time::ptime> {
  // types
  typedef has_difference type;

  // public member functions
   BOOST_STATIC_CONSTANT(bool, value = true);
};
```

Description

`has_difference` public member functions

1.
```
BOOST_STATIC_CONSTANT(bool, value = true);
```

Struct difference_type_of<boost::posix_time::ptime>

boost::icl::difference_type_of<boost::posix_time::ptime>

Synopsis

```
// In header: <boost/icl/ptime.hpp>


struct difference_type_of<boost::posix_time::ptime> {
  // types
  typedef boost::posix_time::time_duration type;
};
```

Struct size_type_of<boost::posix_time::ptime>

boost::icl::size_type_of<boost::posix_time::ptime>

Synopsis

```
// In header: <boost/icl/ptime.hpp>


struct size_type_of<boost::posix_time::ptime> {
  // types
  typedef boost::posix_time::time_duration type;
};
```

Struct is_discrete<boost::posix_time::time_duration>

boost::icl::is_discrete<boost::posix_time::time_duration>

Synopsis

```
// In header: <boost/icl/ptime.hpp>


struct is_discrete<boost::posix_time::time_duration> {
  // types
  typedef is_discrete type;

  // public member functions
   BOOST_STATIC_CONSTANT(bool, value = true);
};
```

Description

`is_discrete` public member functions

1.
```
BOOST_STATIC_CONSTANT(bool, value = true);
```

Struct has_difference<boost::posix_time::time_duration>

boost::icl::has_difference<boost::posix_time::time_duration>

Synopsis

```
// In header: <boost/icl/ptime.hpp>


struct has_difference<boost::posix_time::time_duration> {
  // types
  typedef has_difference type;

  // public member functions
   BOOST_STATIC_CONSTANT(bool, value = true);
};
```

Description

`has_difference` public member functions

1.
```
BOOST_STATIC_CONSTANT(bool, value = true);
```

Struct size_type_of<boost::posix_time::time_duration>

boost::icl::size_type_of<boost::posix_time::time_duration>

Synopsis

```
// In header: <boost/icl/ptime.hpp>


struct size_type_of<boost::posix_time::time_duration> {
  // types
  typedef boost::posix_time::time_duration type;
};
```

Header <boost/icl/rational.hpp>

```
namespace boost {
  namespace icl {
    template<typename Integral> struct is_numeric<boost::rational< Integral >>;
    template<typename Integral> 
      struct is_continuous<boost::rational< Integral >>;
    template<typename Integral> struct is_discrete<boost::rational< Integral >>;
    template<typename Integral> struct has_inverse<boost::rational< Integral >>;
  }
}
```

Struct template is_numeric<boost::rational< Integral >>

boost::icl::is_numeric<boost::rational< Integral >>

Synopsis

```
// In header: <boost/icl/rational.hpp>

template<typename Integral> 
struct is_numeric<boost::rational< Integral >> {
  // types
  typedef is_numeric type;

  // public member functions
   BOOST_STATIC_CONSTANT(bool, value = true);
};
```

Description

`is_numeric` public member functions

1. ```
BOOST_STATIC_CONSTANT(bool, value = true);
```

# Struct template is_continuous<boost::rational< Integral >>

boost::icl::is_continuous<boost::rational< Integral >>

# Synopsis

```
// In header: <boost/icl/rational.hpp>

template<typename Integral>
struct is_continuous<boost::rational< Integral >> {
  // types
  typedef is_continuous type;

  // public member functions
   BOOST_STATIC_CONSTANT(bool, value = true);
};
```

Description

`is_continuous` public member functions

1. ```
BOOST_STATIC_CONSTANT(bool, value = true);
```

Struct template is_discrete<boost::rational< Integral >>

boost::icl::is_discrete<boost::rational< Integral >>

Synopsis

```
// In header: <boost/icl/rational.hpp>

template<typename Integral>
struct is_discrete<boost::rational< Integral >> {
  // types
  typedef is_discrete type;

  // public member functions
   BOOST_STATIC_CONSTANT(bool, value = false);
};
```

Description

`is_discrete` public member functions

1. ```
BOOST_STATIC_CONSTANT(bool, value = false);
```
```

Struct template has_inverse<boost::rational< Integral >>

boost::icl::has_inverse<boost::rational< Integral >>

Synopsis

```
// In header: <boost/icl/rational.hpp>

template<typename Integral>
struct has_inverse<boost::rational< Integral >> {
  // types
  typedef has_inverse type;

  // public member functions
   BOOST_STATIC_CONSTANT(bool, value = (boost::is_signed< Integral >::value));
};
```

Description

`has_inverse` public member functions

1.
```
BOOST_STATIC_CONSTANT(bool, value = (boost::is_signed< Integral >::value));
```

Header <boost/icl/right_open_interval.hpp>

```
namespace boost {
  namespace icl {
    template<typename DomainT, 
             ICL_COMPARE Compare = ICL_COMPARE_INSTANCE(ICL_COMPARE_DEFAULT, DomainT)> 
      class right_open_interval;

    template<typename DomainT, ICL_COMPARE Compare> 
      struct interval_traits<icl::right_open_interval< DomainT, Compare >>;
    template<typename DomainT, ICL_COMPARE Compare> 
      struct interval_bound_type<right_open_interval< DomainT, Compare >>;
    template<typename DomainT, ICL_COMPARE Compare> 
      struct type_to_string<icl::right_open_interval< DomainT, Compare >>;
    template<typename DomainT, ICL_COMPARE Compare> 
      struct value_size<icl::right_open_interval< DomainT, Compare >>;
  }
}
```

Class template right_open_interval

boost::icl::right_open_interval

Synopsis

```
// In header: <boost/icl/right_open_interval.hpp>

template<typename DomainT,
         ICL_COMPARE Compare = ICL_COMPARE_INSTANCE(ICL_COMPARE_DEFAULT, DomainT)>
class right_open_interval {
public:
  // types
  typedef right_open_interval< DomainT, Compare > type;
  typedef DomainT                                 domain_type;

  // construct/copy/destruct
  right_open_interval();
  explicit right_open_interval(const DomainT &);
  right_open_interval(const DomainT &, const DomainT &);

  // public member functions
  typedef ICL_COMPARE_DOMAIN(Compare, DomainT);
  domain_type lower() const;
  domain_type upper() const;
};
```

Description

`right_open_interval` public construct/copy/destruct

1.
```
right_open_interval();
```

Default constructor; yields an empty interval `[0,0)`.

2.
```
explicit right_open_interval(const DomainT & val);
```

Constructor for a singleton interval `[val,val+1)`

3.
```
right_open_interval(const DomainT & low, const DomainT & up);
```

Interval from `low` to `up` with bounds `bounds`

`right_open_interval` public member functions

1.
```
typedef ICL_COMPARE_DOMAIN(Compare, DomainT);
```

2.
```
domain_type lower() const;
```

3.
```
domain_type upper() const;
```

Struct template interval_traits<icl::right_open_interval< DomainT, Compare >>

boost::icl::interval_traits<icl::right_open_interval< DomainT, Compare >>

Synopsis

```
// In header: <boost/icl/right_open_interval.hpp>

template<typename DomainT, ICL_COMPARE Compare>
struct interval_traits<icl::right_open_interval< DomainT, Compare >> {
  // types
  typedef DomainT                                            domain_type;
  typedef icl::right_open_interval< DomainT, Compare > interval_type;

  // public member functions
  typedef ICL_COMPARE_DOMAIN(Compare, DomainT);

  // public static functions
  static interval_type construct(const domain_type &, const domain_type &);
  static domain_type lower(const interval_type &);
  static domain_type upper(const interval_type &);
};
```

Description

`interval_traits` public member functions

1.
```
typedef ICL_COMPARE_DOMAIN(Compare, DomainT);
```

`interval_traits` public static functions

1.
```
static interval_type construct(const domain_type & lo, const domain_type & up);
```

2.
```
static domain_type lower(const interval_type & inter_val);
```

3.
```
static domain_type upper(const interval_type & inter_val);
```

Struct template interval_bound_type<right_open_interval< DomainT, Compare >>

boost::icl::interval_bound_type<right_open_interval< DomainT, Compare >>

Synopsis

```
// In header: <boost/icl/right_open_interval.hpp>

template<typename DomainT, ICL_COMPARE Compare>
struct interval_bound_type<right_open_interval< DomainT, Compare >> {
  // types
  typedef interval_bound_type type;

  // public member functions
   BOOST_STATIC_CONSTANT(bound_type,
                         value = interval_bounds::static_right_open);
};
```

Description

`interval_bound_type` public member functions

1.
```
BOOST_STATIC_CONSTANT(bound_type, value = interval_bounds::static_right_open);
```

Struct template type_to_string<icl::right_open_interval< DomainT, Compare >>

boost::icl::type_to_string<icl::right_open_interval< DomainT, Compare >>

Synopsis

```
// In header: <boost/icl/right_open_interval.hpp>

template<typename DomainT, ICL_COMPARE Compare>
struct type_to_string<icl::right_open_interval< DomainT, Compare >> {

  // public static functions
  static std::string apply();
};
```

Description

`type_to_string` public static functions

1.
```
static std::string apply();
```

Struct template value_size<icl::right_open_interval< DomainT, Compare >>

boost::icl::value_size<icl::right_open_interval< DomainT, Compare >>

Synopsis

```
// In header: <boost/icl/right_open_interval.hpp>

template<typename DomainT, ICL_COMPARE Compare>
struct value_size<icl::right_open_interval< DomainT, Compare >> {

  // public static functions
  static std::size_t apply(const icl::right_open_interval< DomainT > &);
};
```

Description

`value_size` public static functions

1.
```
static std::size_t apply(const icl::right_open_interval< DomainT > &);
```

Header <boost/icl/separate_interval_set.hpp>

```
namespace boost {
  namespace icl {
    template<typename DomainT, 
             ICL_COMPARE Compare = ICL_COMPARE_INSTANCE(ICL_COMPARE_DEFAULT, DomainT), 
           ICL_INTERVAL(ICL_COMPARE) Interval = ICL_INTERVAL_INSTANCE(ICL_INTERVAL_DEFAULT, Do↵
mainT, Compare), 
             ICL_ALLOC Alloc = std::allocator> 
      class separate_interval_set;

    template<typename DomainT, ICL_COMPARE Compare, 
             ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc> 
      struct is_set<icl::separate_interval_set< DomainT, Compare, Interval, Alloc >>;
    template<typename DomainT, ICL_COMPARE Compare, 
             ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc> 
      struct is_interval_container<icl::separate_interval_set< DomainT, Compare, Interval, Al↵
loc >>;
    template<typename DomainT, ICL_COMPARE Compare, 
             ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc> 
      struct is_interval_separator<icl::separate_interval_set< DomainT, Compare, Interval, Al↵
loc >>;
    template<typename DomainT, ICL_COMPARE Compare, 
             ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc> 
      struct type_to_string<icl::separate_interval_set< DomainT, Compare, Interval, Alloc >>;
  }
}
```

Class template separate_interval_set

boost::icl::separate_interval_set — Implements a set as a set of intervals - leaving adjoining intervals separate.

Synopsis

```
// In header: <boost/icl/separate_interval_set.hpp>

template<typename DomainT,
         ICL_COMPARE Compare = ICL_COMPARE_INSTANCE(ICL_COMPARE_DEFAULT, DomainT),
         ICL_INTERVAL(ICL_COMPARE) Interval = ICL_INTERVAL_INSTANCE(ICL_INTERVAL_DEFAULT, Do↵
mainT, Compare),
         ICL_ALLOC Alloc = std::allocator>
class separate_interval_set : public boost::icl::interval_base_set< separate_interval_set< Do↵
mainT, Compare, Interval, Alloc >, DomainT, Compare, Interval, Alloc >
{
public:
  // types
  typedef separate_interval_set< DomainT, Compare, Interval, Alloc >   type;
  typedef interval_base_set< type, DomainT, Compare, Interval, Alloc > base_type;
  typedef type                                                         overloadable_type;
  typedef type                                                         key_object_type;
  typedef interval_set< DomainT, Compare, Interval, Alloc >            joint_type;
  typedef DomainT                                                      domain_type;         ↵
 // The domain type of the set.
  typedef DomainT                                                      codomain_type;       ↵
 // The codomaintype is the same as domain_type.
  typedef DomainT                                                      element_type;        ↵
 // The element type of the set.
  typedef interval_type                                                segment_type;        ↵
 // The segment type of the set.
  typedef exclusive_less_than< interval_type >                         interval_compare;    ↵
 // Comparison functor for intervals.
  typedef exclusive_less_than< interval_type >                         key_compare;         ↵
 // Comparison functor for keys.
  typedef Alloc< interval_type >                                       allocator_type;      ↵
 // The allocator type of the set.
  typedef Alloc< DomainT >                                             domain_allocator_type; ↵
 // allocator type of the corresponding element set
  typedef base_type::atomized_type                                     atomized_type;       ↵
 // The corresponding atomized type representing this interval container of elements.
  typedef base_type::ImplSetT                                          ImplSetT;            ↵
 // Container type for the implementation.
  typedef ImplSetT::key_type                                           key_type;            ↵
 // key type of the implementing container
  typedef ImplSetT::value_type                                         data_type;           ↵
 // data type of the implementing container
  typedef ImplSetT::value_type                                         value_type;          ↵
 // value type of the implementing container
  typedef ImplSetT::iterator                                           iterator;            ↵
 // iterator for iteration over intervals
  typedef ImplSetT::const_iterator                                     const_iterator;      ↵
 // const_iterator for iteration over intervals

  enum @10 { fineness = = 2 };

  // construct/copy/destruct
  separate_interval_set();
  separate_interval_set(const separate_interval_set &);
  template<typename SubType>
    separate_interval_set(const interval_base_set< SubType, DomainT, Compare, Interval, Alloc > &);
  explicit separate_interval_set(const domain_type &);
  explicit separate_interval_set(const interval_type &);
  separate_interval_set(separate_interval_set &&);
  separate_interval_set& operator=(const separate_interval_set &);
  template<typename SubType>
```

```
    separate_interval_set&
    operator=(const interval_base_set< SubType, DomainT, Compare, Interval, Alloc > &);
  separate_interval_set& operator=(separate_interval_set &&);

  // public member functions
  typedef ICL_INTERVAL_TYPE(Interval, DomainT, Compare);
  typedef ICL_COMPARE_DOMAIN(Compare, DomainT);
  template<typename SubType>
    void assign(const interval_base_set< SubType, DomainT, Compare, Interval, Alloc > &);

  // private member functions
  iterator handle_inserted(iterator);
  iterator add_over(const interval_type &, iterator);
  iterator add_over(const interval_type &);
};
```

Description

`separate_interval_set` public construct/copy/destruct

1. ```
   separate_interval_set();
   ```

   Default constructor for the empty object.

2. ```
   separate_interval_set(const separate_interval_set & src);
   ```

   Copy constructor.

3. ```
   template<typename SubType>
     separate_interval_set(const interval_base_set< SubType, DomainT, Compare, Interval, Alloc
   > & src);
   ```

   Copy constructor for base_type.

4. ```
   explicit separate_interval_set(const domain_type & elem);
   ```

   Constructor for a single element.

5. ```
   explicit separate_interval_set(const interval_type & itv);
   ```

   Constructor for a single interval.

6. ```
   separate_interval_set(separate_interval_set && src);
   ```

   Move constructor.

7. ```
   separate_interval_set& operator=(const separate_interval_set & src);
   ```

   Assignment operator.

8. ```
   template<typename SubType>
     separate_interval_set&
     operator=(const interval_base_set< SubType, DomainT, Compare, Interval, Alloc > & src);
   ```

Assignment operator for base type.

9.
```
separate_interval_set& operator=(separate_interval_set && src);
```

Move assignment operator.

`separate_interval_set` public member functions

1.
```
typedef ICL_INTERVAL_TYPE(Interval, DomainT, Compare);
```

The interval type of the set.

2.
```
typedef ICL_COMPARE_DOMAIN(Compare, DomainT);
```

Comparison functor for domain values.

3.
```
template<typename SubType>
  void assign(const interval_base_set< SubType, DomainT, Compare, Interval, Alloc > & src);
```

Assignment from a base `interval_set`.

`separate_interval_set` private member functions

1.
```
iterator handle_inserted(iterator inserted_);
```

2.
```
iterator add_over(const interval_type & addend, iterator last_);
```

3.
```
iterator add_over(const interval_type & addend);
```

Struct template is_set<icl::separate_interval_set< DomainT, Compare, Interval, Alloc >>

boost::icl::is_set<icl::separate_interval_set< DomainT, Compare, Interval, Alloc >>

Synopsis

```
// In header: <boost/icl/separate_interval_set.hpp>

template<typename DomainT, ICL_COMPARE Compare,
         ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc>
struct is_set<icl::separate_interval_set< DomainT, Compare, Interval, Alloc >> {
  // types
  typedef is_set< icl::separate_interval_set< DomainT, Compare, Interval, Alloc > > type;

  // public member functions
   BOOST_STATIC_CONSTANT(bool, value = true);
};
```

Description

`is_set` public member functions

1.
```
BOOST_STATIC_CONSTANT(bool, value = true);
```

Struct template is_interval_container<icl::separate_interval_set< DomainT, Compare, Interval, Alloc >>

boost::icl::is_interval_container<icl::separate_interval_set< DomainT, Compare, Interval, Alloc >>

Synopsis

```
// In header: <boost/icl/separate_interval_set.hpp>

template<typename DomainT, ICL_COMPARE Compare,
         ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc>
struct is_interval_container<icl::separate_interval_set< DomainT, Compare, Interval, Alloc >> {
  // types
  typedef is_interval_container< icl::separate_interval_set< DomainT, Compare, Interval, Al↵
loc > > type;

  // public member functions
   BOOST_STATIC_CONSTANT(bool, value = true);
};
```

Description

`is_interval_container` public member functions

1.
```
BOOST_STATIC_CONSTANT(bool, value = true);
```

Struct template is_interval_separator<icl::separate_interval_set< DomainT, Compare, Interval, Alloc >>

boost::icl::is_interval_separator<icl::separate_interval_set< DomainT, Compare, Interval, Alloc >>

Synopsis

```
// In header: <boost/icl/separate_interval_set.hpp>

template<typename DomainT, ICL_COMPARE Compare,
         ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc>
struct is_interval_separator<icl::separate_interval_set< DomainT, Compare, Interval, Alloc >> {
  // types
  typedef is_interval_separator< icl::separate_interval_set< DomainT, Compare, Interval, Al↵
loc > > type;

  // public member functions
   BOOST_STATIC_CONSTANT(bool, value = true);
};
```

Description

`is_interval_separator` public member functions

1.
```
BOOST_STATIC_CONSTANT(bool, value = true);
```

Struct template type_to_string<icl::separate_interval_set< DomainT, Compare, Interval, Alloc >>

boost::icl::type_to_string<icl::separate_interval_set< DomainT, Compare, Interval, Alloc >>

Synopsis

```
// In header: <boost/icl/separate_interval_set.hpp>

template<typename DomainT, ICL_COMPARE Compare,
         ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc>
struct type_to_string<icl::separate_interval_set< DomainT, Compare, Interval, Alloc >> {

  // public static functions
  static std::string apply();
};
```

Description

`type_to_string` public static functions

1.
```
static std::string apply();
```

Header <boost/icl/split_interval_map.hpp>

```
namespace boost {
  namespace icl {
    template<typename DomainT, typename CodomainT, typename Traits, 
             ICL_COMPARE Compare, ICL_COMBINE Combine, ICL_SECTION Section, 
             ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc> 
      struct is_map<icl::split_interval_map< DomainT, CodomainT, Traits, Compare, Combine, Sec↵
tion, Interval, Alloc >>;
    template<typename DomainT, typename CodomainT, typename Traits, 
             ICL_COMPARE Compare, ICL_COMBINE Combine, ICL_SECTION Section, 
             ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc> 
     struct has_inverse<icl::split_interval_map< DomainT, CodomainT, Traits, Compare, Combine, Sec↵
tion, Interval, Alloc >>;
    template<typename DomainT, typename CodomainT, typename Traits, 
             ICL_COMPARE Compare, ICL_COMBINE Combine, ICL_SECTION Section, 
             ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc> 
      struct is_interval_container<icl::split_interval_map< DomainT, CodomainT, Traits, Com↵
pare, Combine, Section, Interval, Alloc >>;
    template<typename DomainT, typename CodomainT, typename Traits, 
             ICL_COMPARE Compare, ICL_COMBINE Combine, ICL_SECTION Section, 
             ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc> 
     struct is_interval_splitter<icl::split_interval_map< DomainT, CodomainT, Traits, Compare, Com↵
bine, Section, Interval, Alloc >>;
    template<typename DomainT, typename CodomainT, typename Traits, 
             ICL_COMPARE Compare, ICL_COMBINE Combine, ICL_SECTION Section, 
             ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc> 
     struct absorbs_identities<icl::split_interval_map< DomainT, CodomainT, Traits, Compare, Com↵
bine, Section, Interval, Alloc >>;
    template<typename DomainT, typename CodomainT, typename Traits, 
             ICL_COMPARE Compare, ICL_COMBINE Combine, ICL_SECTION Section, 
             ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc> 
     struct is_total<icl::split_interval_map< DomainT, CodomainT, Traits, Compare, Combine, Sec↵
tion, Interval, Alloc >>;
    template<typename DomainT, typename CodomainT, typename Traits, 
             ICL_COMPARE Compare, ICL_COMBINE Combine, ICL_SECTION Section, 
             ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc> 
      struct type_to_string<icl::split_interval_map< DomainT, CodomainT, Traits, Compare, Com↵
bine, Section, Interval, Alloc >>;
  }
}
```

Struct template is_map<icl::split_interval_map< DomainT, CodomainT, Traits, Compare, Combine, Section, Interval, Alloc >>

boost::icl::is_map<icl::split_interval_map< DomainT, CodomainT, Traits, Compare, Combine, Section, Interval, Alloc >>

Synopsis

```
// In header: <boost/icl/split_interval_map.hpp>

template<typename DomainT, typename CodomainT, typename Traits,
         ICL_COMPARE Compare, ICL_COMBINE Combine, ICL_SECTION Section,
         ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc>
struct is_map<icl::split_interval_map< DomainT, CodomainT, Traits, Compare, Combine, Section, In↵
terval, Alloc >> {
  // types
  typedef is_map< icl::split_interval_map< DomainT, CodomainT, Traits, Compare, Combine, Sec↵
tion, Interval, Alloc > > type;

  // public member functions
   BOOST_STATIC_CONSTANT(bool, value = true);
};
```

Description

`is_map` public member functions

1.
```
BOOST_STATIC_CONSTANT(bool, value = true);
```

Struct template has_inverse<icl::split_interval_map< DomainT, CodomainT, Traits, Compare, Combine, Section, Interval, Alloc >>

boost::icl::has_inverse<icl::split_interval_map< DomainT, CodomainT, Traits, Compare, Combine, Section, Interval, Alloc >>

Synopsis

```
// In header: <boost/icl/split_interval_map.hpp>

template<typename DomainT, typename CodomainT, typename Traits,
         ICL_COMPARE Compare, ICL_COMBINE Combine, ICL_SECTION Section,
         ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc>
struct has_inverse<icl::split_interval_map< DomainT, CodomainT, Traits, Compare, Combine, Sec↵
tion, Interval, Alloc >> {
  // types
  typedef has_inverse< icl::split_interval_map< DomainT, CodomainT, Traits, Compare, Combine, Sec↵
tion, Interval, Alloc > > type;

  // public member functions
   BOOST_STATIC_CONSTANT(bool, value = (has_inverse< CodomainT >::value));
};
```

Description

`has_inverse` public member functions

1.
```
BOOST_STATIC_CONSTANT(bool, value = (has_inverse< CodomainT >::value));
```

Struct template is_interval_container<icl::split_interval_map< DomainT, CodomainT, Traits, Compare, Combine, Section, Interval, Alloc >>

boost::icl::is_interval_container<icl::split_interval_map< DomainT, CodomainT, Traits, Compare, Combine, Section, Interval, Alloc >>

Synopsis

```
// In header: <boost/icl/split_interval_map.hpp>

template<typename DomainT, typename CodomainT, typename Traits, 
         ICL_COMPARE Compare, ICL_COMBINE Combine, ICL_SECTION Section, 
         ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc> 
struct is_interval_container<icl::split_interval_map< DomainT, CodomainT, Traits, Compare, Com↵
bine, Section, Interval, Alloc >> {
  // types
  typedef is_interval_container< icl::split_interval_map< DomainT, CodomainT, Traits, Compare, Com↵
bine, Section, Interval, Alloc > > type;

  // public member functions
   BOOST_STATIC_CONSTANT(bool, value = true);
};
```

Description

`is_interval_container` public member functions

1.
```
BOOST_STATIC_CONSTANT(bool, value = true);
```

Struct template is_interval_splitter<icl::split_interval_map< DomainT, CodomainT, Traits, Compare, Combine, Section, Interval, Alloc >>

boost::icl::is_interval_splitter<icl::split_interval_map< DomainT, CodomainT, Traits, Compare, Combine, Section, Interval, Alloc >>

Synopsis

```
// In header: <boost/icl/split_interval_map.hpp>

template<typename DomainT, typename CodomainT, typename Traits, 
         ICL_COMPARE Compare, ICL_COMBINE Combine, ICL_SECTION Section, 
         ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc> 
struct is_interval_splitter<icl::split_interval_map< DomainT, CodomainT, Traits, Compare, Com↵
bine, Section, Interval, Alloc >> {
  // types
  typedef is_interval_splitter< icl::split_interval_map< DomainT, CodomainT, Traits, Compare, Com↵
bine, Section, Interval, Alloc > > type;

  // public member functions
   BOOST_STATIC_CONSTANT(bool, value = true);
};
```

Description

`is_interval_splitter` public member functions

1.
```
BOOST_STATIC_CONSTANT(bool, value = true);
```

Struct template absorbs_identities<icl::split_interval_map< DomainT, CodomainT, Traits, Compare, Combine, Section, Interval, Alloc >>

boost::icl::absorbs_identities<icl::split_interval_map< DomainT, CodomainT, Traits, Compare, Combine, Section, Interval, Alloc >>

Synopsis

```
// In header: <boost/icl/split_interval_map.hpp>

template<typename DomainT, typename CodomainT, typename Traits,
         ICL_COMPARE Compare, ICL_COMBINE Combine, ICL_SECTION Section,
         ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc>
struct absorbs_identities<icl::split_interval_map< DomainT, CodomainT, Traits, Compare, Com↵
bine, Section, Interval, Alloc >> {
  // types
  typedef absorbs_identities< icl::split_interval_map< DomainT, CodomainT, Traits, Compare, Com↵
bine, Section, Interval, Alloc > > type;

  // public member functions
   BOOST_STATIC_CONSTANT(bool, value = (Traits::absorbs_identities));
};
```

Description

`absorbs_identities` public member functions

1.
```
BOOST_STATIC_CONSTANT(bool, value = (Traits::absorbs_identities));
```

Struct template is_total<icl::split_interval_map< DomainT, CodomainT, Traits, Compare, Combine, Section, Interval, Alloc >>

boost::icl::is_total<icl::split_interval_map< DomainT, CodomainT, Traits, Compare, Combine, Section, Interval, Alloc >>

Synopsis

```
// In header: <boost/icl/split_interval_map.hpp>

template<typename DomainT, typename CodomainT, typename Traits, 
         ICL_COMPARE Compare, ICL_COMBINE Combine, ICL_SECTION Section, 
         ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc> 
struct is_total<icl::split_interval_map< DomainT, CodomainT, Traits, Compare, Combine, Section, In↵
terval, Alloc >> {
  // types
  typedef is_total< icl::split_interval_map< DomainT, CodomainT, Traits, Compare, Combine, Sec↵
tion, Interval, Alloc > > type;

  // public member functions
   BOOST_STATIC_CONSTANT(bool, value = (Traits::is_total));
};
```

Description

`is_total` public member functions

1.
```
 BOOST_STATIC_CONSTANT(bool, value = (Traits::is_total));
```

Struct template type_to_string<icl::split_interval_map< DomainT, CodomainT, Traits, Compare, Combine, Section, Interval, Alloc >>

boost::icl::type_to_string<icl::split_interval_map< DomainT, CodomainT, Traits, Compare, Combine, Section, Interval, Alloc >>

Synopsis

```
// In header: <boost/icl/split_interval_map.hpp>

template<typename DomainT, typename CodomainT, typename Traits, 
         ICL_COMPARE Compare, ICL_COMBINE Combine, ICL_SECTION Section, 
         ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc> 
struct type_to_string<icl::split_interval_map< DomainT, CodomainT, Traits, Compare, Combine, Sec↵
tion, Interval, Alloc >> {

  // public static functions
  static std::string apply();
};
```

Description

`type_to_string` public static functions

1.
```
static std::string apply();
```

Header <boost/icl/split_interval_set.hpp>

```
namespace boost {
  namespace icl {
    template<typename DomainT,
             ICL_COMPARE Compare = ICL_COMPARE_INSTANCE(ICL_COMPARE_DEFAULT, DomainT),
           ICL_INTERVAL(ICL_COMPARE) Interval = ICL_INTERVAL_INSTANCE(ICL_INTERVAL_DEFAULT, Do↵
mainT, Compare),
             ICL_ALLOC Alloc = std::allocator>
      class split_interval_set;

    template<typename DomainT, ICL_COMPARE Compare,
             ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc>
      struct is_set<icl::split_interval_set< DomainT, Compare, Interval, Alloc >>;
    template<typename DomainT, ICL_COMPARE Compare,
             ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc>
     struct is_interval_container<icl::split_interval_set< DomainT, Compare, Interval, Alloc >>;
    template<typename DomainT, ICL_COMPARE Compare,
             ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc>
     struct is_interval_splitter<icl::split_interval_set< DomainT, Compare, Interval, Alloc >>;
    template<typename DomainT, ICL_COMPARE Compare,
             ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc>
      struct type_to_string<icl::split_interval_set< DomainT, Compare, Interval, Alloc >>;
  }
}
```

Class template split_interval_set

boost::icl::split_interval_set — implements a set as a set of intervals - on insertion overlapping intervals are split

Synopsis

```
// In header: <boost/icl/split_interval_set.hpp>

template<typename DomainT, 
         ICL_COMPARE Compare = ICL_COMPARE_INSTANCE(ICL_COMPARE_DEFAULT, DomainT), 
         ICL_INTERVAL(ICL_COMPARE) Interval = ICL_INTERVAL_INSTANCE(ICL_INTERVAL_DEFAULT, Do↵
mainT, Compare), 
         ICL_ALLOC Alloc = std::allocator> 
class split_interval_set : public boost::icl::interval_base_set< split_interval_set< DomainT, ↵
Compare, Interval, Alloc >, DomainT, Compare, Interval, Alloc >
{
public:
  // types
  typedef split_interval_set< DomainT, Compare, Interval, Alloc >      type;           
  typedef interval_base_set< type, DomainT, Compare, Interval, Alloc > base_type;      
  typedef interval_set< DomainT, Compare, Interval, Alloc >            joint_type;     
  typedef type                                                         overloadable_type;
  typedef type                                                         key_object_type;
  typedef DomainT                                                     domain_type;          ↵
 // The domain type of the set. 
  typedef DomainT                                                     codomain_type;        ↵
 // The codomaintype is the same as domain_type. 
  typedef DomainT                                                     element_type;         ↵
 // The element type of the set. 
  typedef interval_type                                               segment_type;         ↵
 // The segment type of the set. 
  typedef exclusive_less_than< interval_type >                        interval_compare;     ↵
 // Comparison functor for intervals. 
  typedef exclusive_less_than< interval_type >                        key_compare;          ↵
 // Comparison functor for keys. 
  typedef Alloc< interval_type >                                      allocator_type;       ↵
 // The allocator type of the set. 
  typedef Alloc< DomainT >                                            domain_allocator_type; ↵
 // allocator type of the corresponding element set 
  typedef base_type::atomized_type                                    atomized_type;        ↵
 // The corresponding atomized type representing this interval container of elements. 
  typedef base_type::ImplSetT                                         ImplSetT;             ↵
 // Container type for the implementation. 
  typedef ImplSetT::key_type                                          key_type;             ↵
 // key type of the implementing container 
  typedef ImplSetT::value_type                                        data_type;            ↵
 // data type of the implementing container 
  typedef ImplSetT::value_type                                        value_type;           ↵
 // value type of the implementing container 
  typedef ImplSetT::iterator                                          iterator;             ↵
 // iterator for iteration over intervals 
  typedef ImplSetT::const_iterator                                    const_iterator;       ↵
 // const_iterator for iteration over intervals 

  enum @12 { fineness = = 3 };

  // construct/copy/destruct
  split_interval_set();
  split_interval_set(const split_interval_set &);
  template<typename SubType> 
    split_interval_set(const interval_base_set< SubType, DomainT, Compare, Interval, Alloc > &);
  explicit split_interval_set(const interval_type &);
  explicit split_interval_set(const domain_type &);
  split_interval_set(split_interval_set &&);
  split_interval_set& operator=(const split_interval_set &);
  template<typename SubType>
```

```
    split_interval_set&
    operator=(const interval_base_set< SubType, DomainT, Compare, Interval, Alloc > &);
  split_interval_set& operator=(split_interval_set &&);

  // public member functions
  typedef ICL_INTERVAL_TYPE(Interval, DomainT, Compare);
  typedef ICL_COMPARE_DOMAIN(Compare, DomainT);
  template<typename SubType>
    void assign(const interval_base_set< SubType, DomainT, Compare, Interval, Alloc > &);

  // private member functions
  iterator handle_inserted(iterator);
  iterator add_over(const interval_type &, iterator);
  iterator add_over(const interval_type &);
};
```

Description

`split_interval_set` public construct/copy/destruct

1.
```
split_interval_set();
```

Default constructor for the empty object.

2.
```
split_interval_set(const split_interval_set & src);
```

Copy constructor.

3.
```
template<typename SubType>
  split_interval_set(const interval_base_set< SubType, DomainT, Compare, Interval, Alloc > &
 src);
```

Copy constructor for base_type.

4.
```
explicit split_interval_set(const interval_type & elem);
```

Constructor for a single element.

5.
```
explicit split_interval_set(const domain_type & itv);
```

Constructor for a single interval.

6.
```
split_interval_set(split_interval_set && src);
```

Move constructor.

7.
```
split_interval_set& operator=(const split_interval_set & src);
```

Assignment operator.

8.
```
template<typename SubType>
  split_interval_set&
  operator=(const interval_base_set< SubType, DomainT, Compare, Interval, Alloc > & src);
```

Assignment operator for base type.

9.
```
split_interval_set& operator=(split_interval_set && src);
```

Move assignment operator.

`split_interval_set` public member functions

1.
```
typedef ICL_INTERVAL_TYPE(Interval, DomainT, Compare);
```

The interval type of the set.

2.
```
typedef ICL_COMPARE_DOMAIN(Compare, DomainT);
```

Comparison functor for domain values.

3.
```
template<typename SubType>
  void assign(const interval_base_set< SubType, DomainT, Compare, Interval, Alloc > & src);
```

Assignment from a base `interval_set`.

`split_interval_set` private member functions

1.
```
iterator handle_inserted(iterator inserted_);
```

2.
```
iterator add_over(const interval_type & addend, iterator last_);
```

3.
```
iterator add_over(const interval_type & addend);
```

Struct template is_set<icl::split_interval_set< DomainT, Compare, Interval, Alloc >>

boost::icl::is_set<icl::split_interval_set< DomainT, Compare, Interval, Alloc >>

Synopsis

```
// In header: <boost/icl/split_interval_set.hpp>

template<typename DomainT, ICL_COMPARE Compare,
         ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc>
struct is_set<icl::split_interval_set< DomainT, Compare, Interval, Alloc >> {
  // types
  typedef is_set< icl::split_interval_set< DomainT, Compare, Interval, Alloc > > type;

  // public member functions
   BOOST_STATIC_CONSTANT(bool, value = true);
};
```

Description

`is_set` public member functions

1.
```
BOOST_STATIC_CONSTANT(bool, value = true);
```

Struct template is_interval_container<icl::split_interval_set< DomainT, Compare, Interval, Alloc >>

boost::icl::is_interval_container<icl::split_interval_set< DomainT, Compare, Interval, Alloc >>

Synopsis

```
// In header: <boost/icl/split_interval_set.hpp>

template<typename DomainT, ICL_COMPARE Compare,
         ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc>
struct is_interval_container<icl::split_interval_set< DomainT, Compare, Interval, Alloc >> {
  // types
  typedef is_interval_container< icl::split_interval_set< DomainT, Compare, Interval, Al↵
loc > > type;

  // public member functions
   BOOST_STATIC_CONSTANT(bool, value = true);
};
```

Description

`is_interval_container` public member functions

1.
```
BOOST_STATIC_CONSTANT(bool, value = true);
```

Struct template is_interval_splitter<icl::split_interval_set< DomainT, Compare, Interval, Alloc >>

boost::icl::is_interval_splitter<icl::split_interval_set< DomainT, Compare, Interval, Alloc >>

Synopsis

```
// In header: <boost/icl/split_interval_set.hpp>

template<typename DomainT, ICL_COMPARE Compare,
         ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc>
struct is_interval_splitter<icl::split_interval_set< DomainT, Compare, Interval, Alloc >> {
  // types
  typedef is_interval_splitter< icl::split_interval_set< DomainT, Compare, Interval, Al↵
loc > > type;

  // public member functions
   BOOST_STATIC_CONSTANT(bool, value = true);
};
```

Description

`is_interval_splitter` public member functions

1.
```
BOOST_STATIC_CONSTANT(bool, value = true);
```

Struct template type_to_string<icl::split_interval_set< DomainT, Compare, Interval, Alloc >>

boost::icl::type_to_string<icl::split_interval_set< DomainT, Compare, Interval, Alloc >>

Synopsis

```
// In header: <boost/icl/split_interval_set.hpp>

template<typename DomainT, ICL_COMPARE Compare,
         ICL_INTERVAL(ICL_COMPARE) Interval, ICL_ALLOC Alloc>
struct type_to_string<icl::split_interval_set< DomainT, Compare, Interval, Alloc >> {

  // public static functions
  static std::string apply();
};
```

Description

`type_to_string` public static functions

1.
```
static std::string apply();
```

14:46 15.10.2010