
Geometry

Barend Gehrels

Bruno Lalande

Mateusz Loskot

Copyright © 2009-2012 Barend Gehrels, Bruno Lalande, Mateusz Loskot

Distributed under the Boost Software License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)

Table of Contents

Introduction	2
Quick Start	3
Design Rationale	6
Compilation	17
Indexes	20
Reference matrix	20
Alphabetical Index	24
Reference	29
Access Functions	29
Adapted models	39
Algorithms	69
Arithmetic	179
Concepts	185
Constants	191
Coordinate Systems	193
Core Metafunctions	196
Enumerations	212
Exceptions	213
Iterators	215
Models	216
Strategies	228
Views	248
Release Notes	254
About this documentation	256
Acknowledgments	259

Contributions

Boost.Geometry contains contributions by:

- Akira Takahashi (adaption of Boost.Fusion)
- Alfredo Correa (adaption of Boost.Array)
- Adam Wulkiewicz (spatial indexes) ¹
- Federico Fernández (spatial indexes) ²

¹ Currently an extension

² Currently an extension

Introduction

Boost.Geometry (aka Generic Geometry Library, GGL), part of collection of the Boost C++ Libraries, defines concepts, primitives and algorithms for solving geometry problems.

Boost.Geometry contains a dimension-agnostic, coordinate-system-agnostic and scalable kernel, based on concepts, meta-functions and tag dispatching. On top of that kernel, algorithms are built: area, length, perimeter, centroid, convex hull, intersection (clipping), within (point in polygon), distance, envelope (bounding box), simplify, transform, and much more. The library supports high precision arithmetic numbers, such as [tmath](#).

Boost.Geometry contains instantiable geometry classes, but library users can also use their own. Using registration macros or traits classes their geometries can be adapted to fulfil Boost.Geometry concepts.

Boost.Geometry might be used in all domains where geometry plays a role: mapping and GIS, game development, computer graphics and widgets, robotics, astronomy and more. The core is designed to be as generic as possible and support those domains. For now, the development has been mostly GIS-oriented.

The library follows existing conventions:

- conventions from boost
- conventions from the std library
- conventions and names from one of the [OGC](#) standards on geometry and, more specifically, from the [OGC Simple Feature Specification](#)

The library was first released with Boost 1.47.0 and from that point on it is officially part of the Boost C++ Libraries.

Latest stable version of the source code is included in the [Boost packaged releases](#). It can also be downloaded from the current [Boost release branch](#) in the Boost Subversion repository.

The library development upstream is available from the [Boost trunk](#) in the Boost Subversion repository.

Note that the library **extensions** are not distributed in the official Boost releases, but only available in the [Boost trunk](#) and that they are subject to change.

Boost.Geometry was accepted by Boost at November 28, 2009 ([review report](#)).

There is a Boost.Geometry [mailing list](#). The mailing list and its messages are also accessible from [Nabble](#) as Boost Geometry. Also on the Boost Developers list and on the Boost Users list Boost.Geometry is discussed.

Quick Start

This Quick Start section shows some of the features of Boost.Geometry in the form of annotated, relatively simple, code snippets.

The code below assumes that `boost/geometry.hpp` is included, and that namespace `boost::geometry` is used. Boost.Geometry is header only, so including headerfiles is enough. There is no linking with any library necessary.

```
#include <boost/geometry.hpp>
#include <boost/geometry/geometries/point_xy.hpp>
#include <boost/geometry/geometries/polygon.hpp>

using namespace boost::geometry;
```

Cartesian

It is possible to use only a small part of the library. For example: the distance between two points is a common use case. Boost.Geometry can calculate it from various types. Using one of its own types:

```
model::d2::point_xy<int> p1(1, 1), p2(2, 2);
std::cout << "Distance p1-p2 is: " << distance(p1, p2) << std::endl;
```

If the right headers are included and the types are bound to a coordinate system, various other types can be used as points: plain C array's, Boost.Array's, Boost.Tuple's, Boost.Fusion imported structs, your own classes...

Registering and using a C array:

```
#include <boost/geometry/geometries/adapted/c_array.hpp>

BOOST_GEOMETRY_REGISTER_C_ARRAY_CS(cs::cartesian)
```

```
int a[2] = {1,1};
int b[2] = {2,3};
double d = distance(a, b);
std::cout << "Distance a-b is: " << d << std::endl;
```

Another often used algorithm is point-in-polygon. It is implemented in Boost.Geometry under the name `within`. We show its usage here checking a Boost.Tuple (as a point) located within a polygon, filled with C Array point pairs.

But it is first necessary to register a Boost.Tuple, like the C array:

```
#include <boost/geometry/geometries/adapted/boost_tuple.hpp>

BOOST_GEOMETRY_REGISTER_BOOST_TUPLE_CS(cs::cartesian)
```

```
double points[][2] = {{2.0, 1.3}, {4.1, 3.0}, {5.3, 2.6}, {2.9, 0.7}, {2.0, 1.3}};
model::polygon<model::d2::point_xy<double>> poly;
append(poly, points);
boost::tuple<double, double> p = boost::make_tuple(3.7, 2.0);
std::cout << "Point p is in polygon? " << std::boolalpha << within(p, poly) << std::endl;
```

We can calculate the area of a polygon:

```
std::cout << "Area: " << area(poly) << std::endl;
```

By the nature of a template library, it is possible to mix point types. We calculate distance again, now using a C array point and a Boost.Tuple point:

```
double d2 = distance(a, p);
std::cout << "Distance a-p is: " << d2 << std::endl;
```

The snippets listed above generate the following output:

```
Distance p1-p2 is: 1.41421
Distance a-b is: 2.23607
Point p is in polygon? true
Area: 3.015
Distance a-p is: 2.87924
```

Non-Cartesian

It is also possible to use non-Cartesian points. For example: points on a sphere. When then an algorithm such as distance is used the library "inspects" that it is handling spherical points and calculates the distance over the sphere, instead of applying the Pythagorean theorem.



Note

Boost.Geometry supports a geographical coordinate system, but that is in an extension and not released in the current Boost release.

We approximate the Earth as a sphere and calculate the distance between Amsterdam and Paris:

```
typedef boost::geometry::model::point
<
    double, 2, boost::geometry::cs::spherical_equatorial<boost::geometry::degree>
> spherical_point;

spherical_point amsterdam(4.90, 52.37);
spherical_point paris(2.35, 48.86);

double const earth_radius = 3959; // miles
std::cout << "Distance in miles: " << distance(amsterdam, paris) * earth_radius << std::endl;
```

It writes:

```
Distance in miles: 267.02
```

Adapted structs

Finally an example from a totally different domain: developing window-based applications, for example using QtWidgets. As soon as Qt classes are registered in Boost.Geometry we can use them. We can, for example, check if two rectangles overlap and if so, move the second one to another place:

```
QRect r1(100, 200, 15, 15);
QRect r2(110, 210, 20, 20);
if (overlaps(r1, r2))
{
    assign_values(r2, 200, 300, 220, 320);
}
```

More

In the reference many more examples can be found.

Design Rationale

Suppose you need a C++ program to calculate the distance between two points. You might define a struct:

```
struct mypoint
{
    double x, y;
};
```

and a function, containing the algorithm:

```
double distance(mypoint const& a, mypoint const& b)
{
    double dx = a.x - b.x;
    double dy = a.y - b.y;
    return sqrt(dx * dx + dy * dy);
}
```

Quite simple, and it is usable, but not generic. For a library it has to be designed way further. The design above can only be used for 2D points, for the struct **mypoint** (and no other struct), in a Cartesian coordinate system. A generic library should be able to calculate the distance:

- for any point class or struct, not on just this **mypoint** type
- in more than two dimensions
- for other coordinate systems, e.g. over the earth or on a sphere
- between a point and a line or between other geometry combinations
- in higher precision than *double*
- avoiding the square root: often we don't want to do that because it is a relatively expensive function, and for comparing distances it is not necessary

In this and following sections we will make the design step by step more generic.

Using Templates

The distance function can be changed into a template function. This is trivial and allows calculating the distance between other point types than just **mypoint**. We add two template parameters, allowing input of two different point types.

```
template <typename P1, typename P2>
double distance(P1 const& a, P2 const& b)
{
    double dx = a.x - b.x;
    double dy = a.y - b.y;
    return std::sqrt(dx * dx + dy * dy);
}
```

This template version is slightly better, but not much.

Consider a C++ class where member variables are protected... Such a class does not allow to access *x* and *y* members directly. So, this paragraph is short and we just move on.

Using Traits

We need to take a generic approach and allow any point type as input to the distance function. Instead of accessing `x` and `y` members, we will add a few levels of indirection, using a traits system. The function then becomes:

```
template <typename P1, typename P2>
double distance(P1 const& a, P2 const& b)
{
    double dx = get<0>(a) - get<0>(b);
    double dy = get<1>(a) - get<1>(b);
    return std::sqrt(dx * dx + dy * dy);
}
```

This adapted distance function uses a generic `get` function, with dimension as a template parameter, to access the coordinates of a point. This `get` forwards to the traits system, defined as following:

```
namespace traits
{
    template <typename P, int D>
    struct access {};
}
```

which is then specialized for our **mypoint** type, implementing a static method called `get`:

```
namespace traits
{
    template <>
    struct access<mypoint, 0>
    {
        static double get(mypoint const& p)
        {
            return p.x;
        }
    };
    // same for 1: p.y
    ...
}
```

Calling `traits::access<mypoint, 0>::get(a)` now returns us our `x` coordinate. Nice, isn't it? It is too verbose for a function like this, used so often in the library. We can shorten the syntax by adding an extra free function:

```
template <int D, typename P>
inline double get(P const& p)
{
    return traits::access<P, D>::get(p);
}
```

This enables us to call `get<0>(a)`, for any point having the `traits::access` specialization, as shown in the distance algorithm at the start of this paragraph. So we wanted to enable classes with methods like `x()`, and they are supported as long as there is a specialization of the `access` struct with a static `get` function returning `x()` for dimension 0, and similar for 1 and `y()`.

Dimension Agnosticism

Now we can calculate the distance between points in 2D, points of any structure or class. However, we wanted to have 3D as well. So we have to make it dimension agnostic. This complicates our distance function. We can use a `for` loop to walk through dimensions, but `for` loops have another performance than the straightforward coordinate addition which was there originally. However, we can make more usage of templates and make the distance algorithm as following, more complex but attractive for template fans:

```

template <typename P1, typename P2, int D>
struct pythagoras
{
    static double apply(P1 const& a, P2 const& b)
    {
        double d = get<D-1>(a) - get<D-1>(b);
        return d * d + pythagoras<P1, P2, D-1>::apply(a, b);
    }
};

template <typename P1, typename P2 >
struct pythagoras<P1, P2, 0>
{
    static double apply(P1 const&, P2 const&)
    {
        return 0;
    }
};

```

The distance function is calling that pythagoras structure, specifying the number of dimensions:

```

template <typename P1, typename P2>
double distance(P1 const& a, P2 const& b)
{
    BOOST_STATIC_ASSERT(( dimension<P1>::value == dimension<P2>::value ));

    return sqrt(pythagoras<P1, P2, dimension<P1>::value>::apply(a, b));
}

```

The dimension which is referred to is defined using another traits class:

```

namespace traits
{
    template <typename P>
    struct dimension {};
}

```

which has to be specialized again for the struct mypoint.

Because it only has to publish a value, we conveniently derive it from the Boost.MPL class `boost::mpl::int_`:

```

namespace traits
{
    template <>
    struct dimension<mypoint> : boost::mpl::int_<2>
    {};
}

```

Like the free get function, the library also contains a dimension meta-function.

```

template <typename P>
struct dimension : traits::dimension<P>
{};

```

Below is explained why the extra declaration is useful. Now we have agnosticism in the number of dimensions. Our more generic distance function now accepts points of three or more dimensions. The compile-time assertion will prevent point a having two dimension and point b having three dimensions.

Coordinate Type

We assumed double above. What if our points are in integer?

We can easily add a traits class, and we will do that. However, the distance between two integer coordinates can still be a fractionized value. Besides that, a design goal was to avoid square roots. We handle these cases below, in another paragraph. For the moment we keep returning double, but we allow integer coordinates for our point types. To define the coordinate type, we add another traits class, `coordinate_type`, which should be specialized by the library user:

```
namespace traits
{
    template <typename P>
    struct coordinate_type{};

    // specialization for our mypoint
    template <>
    struct coordinate_type<mypoint>
    {
        typedef double type;
    };
}
```

Like the access function, where we had a free get function, we add a proxy here as well. A longer version is presented later on, the short function would look like this:

```
template <typename P>
struct coordinate_type : traits::coordinate_type<P> {};
```

We now can modify our distance algorithm again. Because it still returns double, we only modify the `pythagoras` computation class. It should return the coordinate type of its input. But, it has two input, possibly different, point types. They might also differ in their coordinate types. Not that that is very likely, but we're designing a generic library and we should handle those strange cases. We have to choose one of the coordinate types and of course we select the one with the highest precision. This is not worked out here, it would be too long, and it is not related to geometry. We just assume that there is a meta-function `select_most_precise` selecting the best type.

So our computation class becomes:

```
template <typename P1, typename P2, int D>
struct pythagoras
{
    typedef typename select_most_precise
        <
            typename coordinate_type<P1>::type,
            typename coordinate_type<P2>::type
        >::type computation_type;

    static computation_type apply(P1 const& a, P2 const& b)
    {
        computation_type d = get<D-1>(a) - get<D-1>(b);
        return d * d + pythagoras <P1, P2, D-1> ::apply(a, b);
    }
};
```

Different Geometries

We have designed a dimension agnostic system supporting any point type of any coordinate type. There are still some tweaks but they will be worked out later. Now we will see how we calculate the distance between a point and a polygon, or between a point and

a line-segment. These formulae are more complex, and the influence on design is even larger. We don't want to add a function with another name:

```
template <typename P, typename S>
double distance_point_segment(P const& p, S const& s)
```

We want to be generic, the distance function has to be called from code not knowing the type of geometry it handles, so it has to be named `distance`. We also cannot create an overload because that would be ambiguous, having the same template signature. There are two solutions:

- tag dispatching
- SFINAE

We select tag dispatching because it fits into the traits system. The earlier versions (previews) of Boost.Geometry used SFINAE but we found it had several drawbacks for such a big design, so the switch to tag dispatching was made.

With tag dispatching the distance algorithm inspects the type of geometry of the input parameters. The distance function will be changed into this:

```
template <typename G1, typename G2>
double distance(G1 const& g1, G2 const& g2)
{
    return dispatch::distance
        <
            typename tag<G1>::type,
            typename tag<G2>::type,
            G1, G2
        >::apply(g1, g2);
}
```

It is referring to the tag meta-function and forwarding the call to the **apply** method of a `dispatch::distance` structure. The **tag** meta-function is another traits class, and should be specialized for per point type, both shown here:

```
namespace traits
{
    template <typename G>
    struct tag {};

    // specialization
    template <>
    struct tag<mypoint>
    {
        typedef point_tag type;
    };
}
```

Free meta-function, like `coordinate_system` and `get`:

```
template <typename G>
struct tag : traits::tag<G> {};
```

Tags (`point_tag`, `segment_tag`, etc) are empty structures with the purpose to specialize a dispatch struct. The dispatch struct for distance, and its specializations, are all defined in a separate namespace and look like the following:

```

namespace dispatch {
    template < typename Tag1, typename Tag2, typename G1, typename G2 >
    struct distance
    {};

    template <typename P1, typename P2>
    struct distance < point_tag, point_tag, P1, P2 >
    {
        static double apply(P1 const& a, P2 const& b)
        {
            // here we call pythagoras
            // exactly like we did before
            ...
        }
    };

    template <typename P, typename S>
    struct distance
    <
        point_tag, segment_tag, P, S
    >
    {
        static double apply(P const& p, S const& s)
        {
            // here we refer to another function
            // implementing point-segment
            // calculations in 2 or 3
            // dimensions...
            ...
        }
    };

    // here we might have many more
    // specializations,
    // for point-polygon, box-circle, etc.
} // namespace

```

So yes, it is possible; the distance algorithm is generic now in the sense that it also supports different geometry types. One drawback: we have to define two dispatch specializations for point - segment and for segment - point separately. That will also be solved, in the paragraph reversibility below. The example below shows where we are now: different point types, geometry types, dimensions.

```

point a(1,1);
point b(2,2);
std::cout << distance(a,b) << std::endl;
segment s1(0,0,5,3);
std::cout << distance(a, s1) << std::endl;
rgb red(255, 0, 0);
rbc orange(255, 128, 0);
std::cout << "color distance: " << distance(red, orange) << std::endl;

```

Kernel Revisited

We described above that we had a traits class `coordinate_type`, defined in namespace `traits`, and defined a separate `coordinate_type` class as well. This was actually not really necessary before, because the only difference was the namespace clause. But now that we have another geometry type, a segment in this case, it is essential. We can call the `coordinate_type` meta-function for any geometry type, point, segment, polygon, etc, implemented again by tag dispatching:

```
template <typename G>
struct coordinate_type
{
    typedef typename dispatch::coordinate_type
        <
            typename tag<G>::type, G
        >::type type;
};
```

Inside the dispatch namespace this meta-function is implemented twice: a generic version and one specialization for points. The specialization for points calls the traits class. The generic version calls the point specialization, as a sort of recursive meta-function definition:

```
namespace dispatch
{
    // Version for any geometry:
    template <typename GeometryTag, typename G>
    struct coordinate_type
    {
        typedef typename point_type
            <
                GeometryTag, G
            >::type point_type;

        // Call specialization on point-tag
        typedef typename coordinate_type < point_tag, point_type >::type type;
    };

    // Specialization for point-type:
    template <typename P>
    struct coordinate_type<point_tag, P>
    {
        typedef typename
            traits::coordinate_type<P>::type
            type;
    };
}
```

So it calls another meta-function `point_type`. This is not elaborated in here but realize that it is available for all geometry types, and typedefs the point type which makes up the geometry, calling it type.

The same applies for the meta-function `dimension` and for the upcoming meta-function `coordinate system`.

Coordinate System

Until here we assumed a Cartesian system. But we know that the Earth is not flat. Calculating a distance between two GPS-points with the system above would result in nonsense. So we again extend our design. We define for each point type a coordinate system type using the traits system again. Then we make the calculation dependant on that coordinate system.

Coordinate system is similar to coordinate type, a meta-function, calling a dispatch function to have it for any geometry-type, forwarding to its point specialization, and finally calling a traits class, defining a typedef type with a coordinate system. We don't show that all here again. We only show the definition of a few coordinate systems:

```
struct cartesian {};

template<typename DegreeOrRadian>
struct geographic
{
    typedef DegreeOrRadian units;
};
```

So Cartesian is simple, for geographic we can also select if its coordinates are stored in degrees or in radians.

The distance function will now change: it will select the computation method for the corresponding coordinate system and then call the dispatch struct for distance. We call the computation method specialized for coordinate systems a strategy. So the new version of the distance function is:

```
template <typename G1, typename G2>
double distance(G1 const& g1, G2 const& g2)
{
    typedef typename strategy_distance
        <
            typename coordinate_system<G1>::type,
            typename coordinate_system<G2>::type,
            typename point_type<G1>::type,
            typename point_type<G2>::type,
            dimension<G1>::value
        >::type strategy;

    return dispatch::distance
        <
            typename tag<G1>::type,
            typename tag<G2>::type,
            G1, G2, strategy
        >::apply(g1, g2, strategy());
}
```

The strategy_distance mentioned here is a struct with specializations for different coordinate systems.

```
template <typename T1, typename T2, typename P1, typename P2, int D>
struct strategy_distance
{
    typedef void type;
};

template <typename P1, typename P2, int D>
struct strategy_distance<cartesian, cartesian, P1, P2, D>
{
    typedef pythagoras<P1, P2, D> type;
};
```

So, here is our pythagoras again, now defined as a strategy. The distance dispatch function just calls its apply method.

So this is an important step: for spherical or geographical coordinate systems, another strategy (computation method) can be implemented. For spherical coordinate systems have the haversine formula. So the dispatching traits struct is specialized like this

```
template <typename P1, typename P2, int D = 2>
struct strategy_distance<spherical, spherical, P1, P2, D>
{
    typedef haversine<P1, P2> type;
};

// struct haversine with apply function
// is omitted here
```

For geography, we have some alternatives for distance calculation. There is the Andoyer method, fast and precise, and there is the Vincenty method, slower and more precise, and there are some less precise approaches as well.

Per coordinate system, one strategy is defined as the default strategy. To be able to use another strategy as well, we modify our design again and add an overload for the distance algorithm, taking a strategy object as a third parameter.

This new overload distance function also has the advantage that the strategy can be constructed outside the distance function. Because it was constructed inside above, it could not have construction parameters. But for Andoyer or Vincenty, or the haversine formula, it certainly makes sense to have a constructor taking the radius of the earth as a parameter.

So, the distance overloaded function is:

```
template <typename G1, typename G2, typename S>
double distance(G1 const& g1, G2 const& g2, S const& strategy)
{
    return dispatch::distance
        <
            typename tag<G1>::type,
            typename tag<G2>::type,
            G1, G2, S
        >::apply(g1, g2, strategy);
}
```

The strategy has to have a method `apply` taking two points as arguments (for points). It is not required that it is a static method. A strategy might define a constructor, where a configuration value is passed and stored as a member variable. In those cases a static method would be inconvenient. It can be implemented as a normal method (with the `const` qualifier).

We do not list all implementations here, Vincenty would cover half a page of mathematics, but you will understand the idea. We can call distance like this:

```
distance(c1, c2)
```

where `c1` and `c2` are Cartesian points, or like this:

```
distance(g1, g2)
```

where `g1` and `g2` are Geographic points, calling the default strategy for Geographic points (e.g. Andoyer), and like this:

```
distance(g1, g2, vincenty<G1, G2>(6275))
```

where a strategy is specified explicitly and constructed with a radius.

Point Concept

The five traits classes mentioned in the previous sections form together the Point Concept. Any point type for which specializations are implemented in the traits namespace should be accepted as a valid type. So the Point Concept consists of:

- a specialization for `traits::tag`

- a specialization for `traits::coordinate_system`
- a specialization for `traits::coordinate_type`
- a specialization for `traits::dimension`
- a specialization for `traits::access`

The last one is a class, containing the method `get` and the (optional) method `set`, the first four are metafunctions, either defining type or declaring a value (conform MPL conventions).

So we now have agnosticism for the number of dimensions, agnosticism for coordinate systems; the design can handle any coordinate type, and it can handle different geometry types. Furthermore we can specify our own strategies, the code will not compile in case of two points with different dimensions (because of the assertion), and it will not compile for two points with different coordinate systems (because there is no specialization). A library can check if a point type fulfills the requirements imposed by the concepts. This is handled in the upcoming section Concept Checking.

Return Type

We promised that calling `std::sqrt` was not always necessary. So we define a distance result struct that contains the squared value and is convertible to a double value. This, however, only has to be done for `pythagoras`. The spherical distance functions do not take the square root so for them it is not necessary to avoid the expensive square root call; they can just return their distance.

So the distance result struct is dependant on strategy, therefore made a member type of the strategy. The result struct looks like this:

```
template<typename T = double>
struct cartesian_distance
{
    T sq;
    explicit cartesian_distance(T const& v) : sq (v) {}

    inline operator T() const
    {
        return std::sqrt(sq);
    }
};
```

It also has operators defined to compare itself to other results without taking the square root.

Each strategy should define its return type, within the strategy class, for example:

```
typedef cartesian_distance<T> return_type;
```

or:

```
typedef double return_type;
```

for cartesian (pythagoras) and spherical, respectively.

Again our distance function will be modified, as expected, to reflect the new return type. For the overload with a strategy it is not complex:

```
template < typename G1, typename G2, typename Strategy >
typename Strategy::return_type distance( G1 const& G1 , G2 const& G2 , S const& strategy)
```

But for the one without strategy we have to select strategy, coordinate type, etc. It would be spacious to do it in one line so we add a separate meta-function:

```
template <typename G1, typename G2 = G1>
struct distance_result
{
    typedef typename point_type<G1>::type P1;
    typedef typename point_type<G2>::type P2;
    typedef typename strategy_distance
        <
            typename cs_tag<P1>::type,
            typename cs_tag<P2>::type,
            P1, P2
        >::type S;

    typedef typename S::return_type type;
};
```

and modify our distance function:

```
template <typename G1, typename G2>
inline typename distance_result<G1, G2>::type distance(G1 const& g1, G2 const& g2)
{
    // ...
}
```

Of course also the apply functions in the dispatch specializations will return a result like this. They have a strategy as a template parameter everywhere, making the less verbose version possible.

Summary

In this design rationale, Boost.Geometry is step by step designed using tag dispatching, concepts, traits, and metaprogramming. We used the well-known distance function to show the design.

Boost.Geometry is designed like described here, with some more techniques as automatically reversing template arguments, tag casting, and reusing implementation classes or dispatch classes as policies in other dispatch classes.

Compilation

Boost.Geometry is a headers-only library. Users only need to include the library headers in their programs in order to be able to access definitions and algorithms provided by the Boost.Geometry library. No linking against any binaries is required.

Boost.Geometry is only dependant on headers-only Boost libraries. It does not introduce indirect dependencies on any binary libraries.

In order to be able to use Boost.Geometry, the only thing users need to do is to download and/or install Boost and specify location to include directories, so `include` directives of this scheme will work:

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

Supported Compilers

Boost.Geometry library has been successfully tested with the following compilers:

- MSVC (including Express Editions)
 - 10.0 (MSVC 2010)
 - 9.0 (MSVC 2008)
 - 8.0 (MSVC 2005)
- gcc
 - gcc 4.7.0
 - gcc 4.6.2
 - gcc 4.6.1
 - gcc 4.6.0
 - gcc 4.5.2
 - gcc 4.4.0
 - gcc 4.3.4
 - gcc 4.2.1
 - gcc 3.4.6
- clang
 - clang x.x
- darwin
 - darwin 4.0.1
 - darwin 4.4
- intel
 - intel 11.1
 - intel 11.0

- intel 10.1
- pathscale
 - pathscale 4.0.8

Boost.Geometry uses Boost.Build, a text-based system for developing and testing software, to configure, build and execute unit tests and example programs. The build configuration is provided as a collection of `Jamfile.v2` files.

For gcc, flag `-Wno-long-long` can be used to surpress some warnings originating from Boost.

Includes

The most convenient headerfile including all algorithms and strategies is `geometry.hpp`:

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

This is the main header of the Boost.Geometry library and it is recommended to include this file.

Alternatively, it is possible to include Boost.Geometry header files separately. However, this may be inconvenient as header files might be renamed or moved occasionally in future.

Another often used header is `geometries.hpp`:

```
#include <boost/geometry/geometries/geometries.hpp>
```

This includes definitions of all provided geometry types: point, linestring, polygon, ring, box. The file `geometries.hpp` is not included in the `geometry.hpp` headerfile because users should be given the liberty to use their own geometries and not the provided ones. However, for the Boost.Geometry users who want to use the provided geometries it is useful to include.

For users using multi-geometries:

```
#include <boost/geometry/multi/geometries/multi_geometries.hpp>
```

Advanced Includes

Users who have their own geometries and want to use algorithms from Boost.Geometry might include the files containing registration macro's, like:

```
#include <boost/geometry/geometries/register/point.hpp>
```

Performance

The enumeration below is not exhaustive but can contain hints to improve the performance:

- For Microsoft MSVC, set define `_SECURE_SCL=0` for preprocessor.
- For Microsoft MSVC, set define `_HAS_ITERATOR_DEBUGGING=0` for preprocessor.
- Use of [STLport](http://ericniebler.com/STLport/), a popular open-source implementation of the STL, may result in significantly faster code than use of the C++ standard library provided by MSVC.
- Turn on compiler optimizations, compile in release mode.

Problems with Intellisense

Both versions of MSVC, 2005 and 2008 (including Express Editions) can hang trying to resolve symbols and give [IntelliSense](#) suggestions while typing in a bracket or angle bracket. This is not directly related to Boost.Geometry, but is caused by problems with handling by this IDE large C++ code base with intensively used templates, such as Boost and Boost.Geometry. If this is inconvenient, IntelliSense can be turned off:

“(...)disabling IntelliSense in VC++. There is a file called `feacp.dll` in `<VS8INSTALL>/VC/vcpackages` folder. Renaming this file will disable Intellisense feature.”

-- [Intellisense issues in Visual C++ 2005](#)

Indexes

Reference matrix

Geometry Concepts

0-dimensional

Point
MultiPoint

1-dimensional

Segment
Linestring
MultiLinestring

2-dimensional

Box
Ring
Polygon
MultiPolygon

Geometry Models

0-dimensional

point
point_xy
multi_point

1-dimensional

linestring
multi_linestring
segment
referring_segment

2-dimensional

box
ring
polygon
multi_polygon

0-dimensional (adapted)

Boost.Array
Boost.Fusion
Boost.Polygon's point_data
Boost.Tuple
C arrays

1-dimensional (adapted)

2-dimensional (adapted)

Boost.Polygon's rectangle_data
Boost.Polygon's polygon_data
Boost.Polygon's polygon_with_holes_data

0-dimensional (macro's for adaption)

BOOST_GEOMETRY_REGISTER_POINT_2D
BOOST_GEOMETRY_REGISTER_POINT_2D_CONST
BOOST_GEOMETRY_REGISTER_POINT_2D_GET_SET
BOOST_GEOMETRY_REGISTER_POINT_3D
BOOST_GEOMETRY_REGISTER_POINT_3D_CONST
BOOST_GEOMETRY_REGISTER_POINT_3D_GET_SET
BOOST_GEOMETRY_REGISTER_MULTI_POINT
BOOST_GEOMETRY_REGISTER_MULTI_POINT_TEMPLATED

1-dimensional (macro's for adaption)

BOOST_GEOMETRY_REGISTER_LINESTRING
BOOST_GEOMETRY_REGISTER_LINESTRING_TEMPLATED
BOOST_GEOMETRY_REGISTER_MULTI_LINESTRING
BOOST_GEOMETRY_REGISTER_MULTI_LINESTRING_TEMPLATED

2-dimensional (macro's for adaption)

BOOST_GEOMETRY_REGISTER_BOX
BOOST_GEOMETRY_REGISTER_BOX_2D_4VALUES
BOOST_GEOMETRY_REGISTER_BOX_TEMPLATED
BOOST_GEOMETRY_REGISTER_RING
BOOST_GEOMETRY_REGISTER_RING_TEMPLATED
BOOST_GEOMETRY_REGISTER_MULTI_POLYGON
BOOST_GEOMETRY_REGISTER_MULTI_POLYGON_TEMPLATED

Core

Metafunctions

[cs_tag](#)
[closure](#)
[coordinate_type](#)
[coordinate_system](#)
[dimension](#)
[interior_type](#)
[is_radian](#)
[point_order](#)
[point_type](#)
[ring_type](#)
[tag](#)
[tag_cast](#)

Access Functions

[get](#)
[set](#)
[exterior_ring](#)
[interior_rings](#)

Classes

[exception](#)
[centroid_exception](#)

Constants

Numeric

[max_corner](#)
[min_corner](#)
[order_selector](#)
[closure_selector](#)

Types

[degree](#)
[radian](#)

Coordinate Systems

Classes

[cs::cartesian](#)
[cs::spherical](#)
[cs::spherical_equatorial](#)
[cs::geographic](#)

Iterators

[closing_iterator](#)
[ever_circling_iterator](#)

Views

[box_view](#)
[segment_view](#)
[closeable_view](#)
[reversible_view](#)
[identity_view](#)

Algorithms

Geometry Constructors

[make](#)
[make_inverse](#)
[make_zero](#)

Predicates

[covered_by](#)
[disjoint](#)
[equals](#)
[intersects](#)
[overlaps](#)
[within](#)

Append

[append](#)

Area

[area](#)

Assign

[assign](#)
[assign_inverse](#)
[assign_zero](#)
[assign_points](#)
[assign_values](#) ([2](#) [3](#) [4](#) coordinate values)

Centroid

[centroid](#)

Clear

[clear](#)

Convert

[convert](#)

Convex Hull

[convex_hull](#)

Correct

[correct](#)

Distance

[distance](#)

Difference

[difference](#)
[sym_difference](#)

Envelope

[envelope](#)

Expand

[expand](#)

For Each

[for each](#) (point, segment)

Intersection

[intersection](#)

Length

[length](#)

Num_ (counting)

[num_interior_rings](#)
[num_geometries](#)
[num_points](#)

Perimeter

[perimeter](#)

Reverse

[reverse](#)

Simplify

[simplify](#)

Transform

[transform](#)

Union

[union](#)

Unique

[unique](#)

Strategies

Area

strategy::area::surveyor
strategy::area::huiller

Centroid

strategy::centroid::bashein_detmer
strategy::centroid::centroid_average

Distance

strategy::distance::projected_point
strategy::distance::pythagoras
strategy::distance::cross_track
strategy::distance::haversine

Convex Hull

strategy::convex_hull::graham_andrew

Side

strategy::side::side_by_triangle
strategy::side::side_by_cross_track
strategy::side::spherical_side_formula

Simplify

strategy::simplify::douglas_peucker

Transform

strategy::transform::inverse_transformer
strategy::transform::map_transformer
strategy::transform::ublas_transformer
strategy::transform::translate_transformer
strategy::transform::scale_transformer
strategy::transform::rotate_transformer

Within

strategy::winding
strategy::crossings_multiply
strategy::franklin

Arithmetic

Add

add_point
add_value

Subtract

subtract_point
subtract_value

Multiply

multiply_point
multiply_value

Divide

divide_point
divide_value

Products

dot_product

Alphabetical Index

A

add_point, 179
add_value, 180
append, 84
area, 69, 72, 233, 234
assign, 74
assign_inverse, 76
assign_point, 180
assign_points, 78
assign_value, 181
assign_values, 80, 81, 82

assign_zero, 83
average, 235

B

bashein_detmer, 236
BOOST_GEOMETRY_REGISTER_BOX, 51
BOOST_GEOMETRY_REGISTER_BOX_2D_4VALUES, 52
BOOST_GEOMETRY_REGISTER_BOX_TEMPLATED, 54
BOOST_GEOMETRY_REGISTER_LINESTRING, 55
BOOST_GEOMETRY_REGISTER_LINESTRING_TEMPLATED, 56
BOOST_GEOMETRY_REGISTER_MULTI_LINESTRING, 57
BOOST_GEOMETRY_REGISTER_MULTI_LINESTRING_TEMPLATED, 58
BOOST_GEOMETRY_REGISTER_MULTI_POINT, 58
BOOST_GEOMETRY_REGISTER_MULTI_POINT_TEMPLATED, 59
BOOST_GEOMETRY_REGISTER_MULTI_POLYGON, 60
BOOST_GEOMETRY_REGISTER_MULTI_POLYGON_TEMPLATED, 61
BOOST_GEOMETRY_REGISTER_POINT_2D, 62
BOOST_GEOMETRY_REGISTER_POINT_2D_CONST, 63
BOOST_GEOMETRY_REGISTER_POINT_2D_GET_SET, 64
BOOST_GEOMETRY_REGISTER_POINT_3D, 65
BOOST_GEOMETRY_REGISTER_POINT_3D_CONST, 65
BOOST_GEOMETRY_REGISTER_POINT_3D_GET_SET, 66
BOOST_GEOMETRY_REGISTER_RING, 67
BOOST_GEOMETRY_REGISTER_RING_TEMPLATED, 68
box, 224
box_view, 248
buffer, 86

C

cartesian, 193
centroid, 88, 90, 235, 236
centroid_exception, 214
clear, 96
clockwise, 213
closeable_view, 251
closed, 212
closing_iterator, 215
closure, 196
closure_selector, 212
closure_underterminated, 212
comparable_distance, 116
convert, 99
convex_hull, 102, 237
coordinate_system, 197
coordinate_type, 198
correct, 105
counterclockwise, 213
covered_by, 107, 109
crossings_multiply, 247
cross_track, 231
cs, 193, 194, 194, 195
cs_tag, 199

D

d2, 218
degree, 199
difference, 111

dimension, 200
disjoint, 115
distance, 118, 120, 228, 229, 230, 231
divide_point, 181
divide_value, 182
dot_product, 182
douglas_peucker, 240

E

envelope, 121
equals, 126
ever_circling_iterator, 215
exception, 213
expand, 128
exterior_ring, 37, 37

F

for_each_point, 130, 131
for_each_segment, 133, 136
franklin, 246

G

geographic, 195
get, 29, 30
get_as_radian, 32
graham_andrew, 237

H

haversine, 229
huiller, 234

I

identity_view, 252
interior_rings, 38, 38
interior_type, 201
intersection, 136
intersects, 139, 141
inverse_transformer, 241
is_radian, 202

L

length, 142, 143
linestring, 220

M

make, 145, 147
make_inverse, 148
make_zero, 149
map_transformer, 242
model, 216, 218, 220, 221, 222, 223, 224, 224, 226, 227, 227
multiply_point, 183
multiply_value, 183
multi_linestring, 223
multi_point, 222
multi_polygon, 224

N

num_geometries, 150
 num_interior_rings, 151
 num_points, 153

O

open, 212
 order_selector, 213
 order_undetermined, 213
 overlaps, 155

P

perimeter, 155, 156
 point, 216
 point_order, 203
 point_type, 204
 point_xy, 218
 polygon, 221
 projected_point, 230
 pythagoras, 228

R

radian, 205
 referring_segment, 227
 return_buffer, 87
 return_centroid, 93, 94
 return_envelope, 123
 reverse, 157
 reversible_view, 252
 ring, 226
 ring_type, 206
 rotate_transformer, 243

S

scale_transformer, 243
 segment, 227
 segment_view, 250
 set, 33, 34
 set_from_radian, 36
 side, 238, 239, 239
 side_by_cross_track, 239
 side_by_triangle, 238
 simplify, 159, 160, 240
 spherical, 194
 spherical_equatorial, 194
 spherical_side_formula, 239
 strategy, 228, 229, 230, 231, 233, 234, 235, 236, 237, 238, 239, 239, 240, 241, 242, 243, 243, 244, 244, 245, 246, 247
 subtract_point, 184
 subtract_value, 184
 surveyor, 233
 sym_difference, 163

T

tag, 207
 tag_cast, 210
 transform, 165, 167, 241, 242, 243, 243, 244, 244

translate_transformer, 244

U

ublas_transformer, 244

union_, 169

unique, 172

W

winding, 245

within, 174, 177, 245, 246, 247

Reference

Access Functions

get

get

Get coordinate value of a geometry (usually a point)

Description

The free functions **get** and **set** are two of the most important functions of Boost.Geometry, both within the library, as also for the library user. With these two functions you normally get and set coordinate values from and for a point, box, segment or sphere.

Synopsis

```
template<std::size_t Dimension, typename Geometry>
coordinate_type<Geometry>::type get(Geometry const & geometry)
```

Parameters

Type	Concept	Name	Description
Dimension	Dimension, this template parameter is required. Should contain [0 .. n-1] for an n-dimensional geometry	-	Must be specified
Geometry const &	Any type fulfilling a Geometry Concept (usually a Point Concept)	geometry	A model of the specified concept (usually a point)

Returns

The coordinate value of specified dimension of specified geometry

Behavior

Case	Behavior
Point	Returns the coordinate of a point
Circle or Sphere	Returns the coordinate of the center of a circle or sphere (currently in an extension)
Spherical	Returns the coordinate of a point, in either Radian's or Degree's, depending on specified units

Complexity

Constant

Example

Get the coordinate of a point

```
#include <iostream>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/point_xy.hpp>

namespace bg = boost::geometry;

int main()
{
    bg::model::d2::point_xy<double> point(1, 2);

    double x = bg::get<0>(point);
    double y = bg::get<1>(point);

    std::cout << "x=" << x << " y=" << y << std::endl;

    return 0;
}
```

Output:

```
x=1 y=2
```

Header

Either

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

Or

```
#include <boost/geometry/core/access.hpp>
```

get (with index)

get coordinate value of a Box or Segment

Description

The free functions **get** and **set** are two of the most important functions of Boost.Geometry, both within the library, as also for the library user. With these two functions you normally get and set coordinate values from and for a point, box, segment or sphere.

Synopsis

```
template<std::size_t Index, std::size_t Dimension, typename Geometry>
coordinate_type<Geometry>::type get(Geometry const & geometry)
```

Parameters

Type	Concept	Name	Description
Index	Index, this template parameter is required. For a Box: either <code>min_corner</code> or <code>max_corner</code> . For a Segment: either 0 or 1 for first or last point.	-	Must be specified
Dimension	Dimension, this template parameter is required. Should contain [0 .. n-1] for an n-dimensional geometry	-	Must be specified
Geometry const &	Any type fulfilling a Box Concept or a Segment Concept	geometry	A model of the specified concept

Returns

coordinate value

Header

Either

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

Or

```
#include <boost/geometry/core/access.hpp>
```

Behavior

Case	Behavior
Rectangle	Returns the coordinate of a box (use <code>min_corner</code> , <code>max_corner</code> to specify which of the points to get)
Segment	Returns the coordinate of a segment (use 0, 1 to specify which of the two points to get)

Complexity

Constant

Example

Get the coordinate of a box

```

#include <iostream>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/point_xy.hpp>

namespace bg = boost::geometry;

int main()
{
    bg::model::box<bg::model::d2::point_xy<double> > box;

    bg::assign_values(box, 1, 3, 5, 6);

    std::cout << "Box: "
        << " " << bg::get<bg::min_corner, 0>(box)
        << " " << bg::get<bg::min_corner, 1>(box)
        << " " << bg::get<bg::max_corner, 0>(box)
        << " " << bg::get<bg::max_corner, 1>(box)
        << std::endl;

    return 0;
}

```

Output:

```
Box: 1 3 5 6
```

get_as_radian

get coordinate value of a point, result is in Radian

Description

Result is in Radian, even if source coordinate system is in Degrees

Synopsis

```

template<std::size_t Dimension, typename Geometry>
fp_coordinate_type<Geometry>::type get_as_radian(Geometry const & geometry)

```

Parameters

Type	Concept	Name	Description
Dimension	dimension	-	Must be specified
Geometry const &	geometry	geometry	geometry to get coordinate value from

Returns

coordinate value

Header

Either

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


Or

```
#include <boost/geometry/core/radian_access.hpp>
```

set

set

Set coordinate value of a geometry (usually a point)

Description

The free functions **get** and **set** are two of the most important functions of Boost.Geometry, both within the library, as also for the library user. With these two functions you normally get and set coordinate values from and for a point, box, segment or sphere.

Synopsis

```
template<std::size_t Dimension, typename Geometry>
void set(Geometry & geometry, typename coordinate_type< Geometry >::type const & value)
```

Parameters

Type	Concept	Name	Description
Dimension	Dimension, this template parameter is required. Should contain [0 .. n-1] for an n-dimensional geometry	-	Must be specified
Geometry &	Any type fulfilling a Geometry Concept (usually a Point Concept)	geometry	A model of the specified concept (usually a point)
typename coordinate_type< Geometry >::type const &		value	The coordinate value to set

Header

Either

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

Or

```
#include <boost/geometry/core/access.hpp>
```



Note

If you host both the `std::` library namespace and `boost::geometry::` namespace `set` might become ambiguous, `std::set` is a collection. So don't do that or refer to `geometry::set` then explicitly.

Behavior

Case	Behavior
Point	Sets the coordinate of a point
Circle or Sphere	Sets the coordinate of the center of a circle or sphere (currently in an extension)
Spherical	Sets the coordinate of a point, in either Radian's or Degree's, depending on specified units

Complexity

Constant

Example

Set the coordinate of a point

```
#include <iostream>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/point_xy.hpp>

namespace bg = boost::geometry;

int main()
{
    bg::model::d2::point_xy<double> point;

    bg::set<0>(point, 1);
    bg::set<1>(point, 2);

    std::cout << "Location: " << bg::dsv(point) << std::endl;

    return 0;
}
```

Output:

```
Location: (1, 2)
```

set (with index)

set coordinate value of a Box / Segment

Description

The free functions **get** and **set** are two of the most important functions of Boost.Geometry, both within the library, as also for the library user. With these two functions you normally get and set coordinate values from and for a point, box, segment or sphere.

Synopsis

```
template<std::size_t Index, std::size_t Dimension, typename Geometry>
void set(Geometry & geometry, typename coordinate_type< Geometry >::type const & value)
```

Parameters

Type	Concept	Name	Description
Index	Index, this template parameter is required. For a Box: either <code>min_corner</code> or <code>max_corner</code> . For a Segment: either 0 or 1 for first or last point.	-	Must be specified
Dimension	Dimension, this template parameter is required. Should contain [0 .. n-1] for an n-dimensional geometry	-	Must be specified
Geometry &	Any type fulfilling a Box Concept or a Segment Concept	geometry	A model of the specified concept
typename coordinate_type< Geometry >::type const &		value	The coordinate value to set

Header

Either

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

Or

```
#include <boost/geometry/core/access.hpp>
```

Behavior

Case	Behavior
Rectangle	Sets the coordinate of a box (use <code>min_corner</code> , <code>max_corner</code> to specify which of the points to set)
Segment	Sets the coordinate of a segment (use 0, 1 to specify which of the two points to set)

Complexity

Constant

Example

Set the coordinate of a box

```

#include <iostream>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/point_xy.hpp>

namespace bg = boost::geometry;

int main()
{
    bg::model::box<bg::model::d2::point_xy<double> > box;

    bg::set<bg::min_corner, 0>(box, 0);
    bg::set<bg::min_corner, 1>(box, 2);
    bg::set<bg::max_corner, 0>(box, 4);
    bg::set<bg::max_corner, 1>(box, 5);

    std::cout << "Extent: " << bg::dsv(box) << std::endl;

    return 0;
}

```

Output:

```
Extent: ((0, 2), (4, 5))
```

set_from_radian

set coordinate value (in radian) to a point

Description

Coordinate value will be set correctly, if coordinate system of point is in Degree, Radian value will be converted to Degree

Synopsis

```

template<std::size_t Dimension, typename Geometry>
void set_from_radian(Geometry & geometry, typename fp_coordinate_type< Geometry >::type const & radians)

```

Parameters

Type	Concept	Name	Description
Dimension	dimension	-	Must be specified
Geometry &	geometry	geometry	geometry to assign coordinate to
typename fp_coordinate_type< Geometry >::type const &		radians	coordinate value to assign

Header

Either

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

Or

```
#include <boost/geometry/core/radian_access.hpp>
```

exterior_ring

exterior_ring

Function to get the exterior_ring ring of a polygon.

Synopsis

```
template<typename Polygon, >
ring_return_type<Polygon>::type exterior_ring(Polygon & polygon)
```

Parameters

Type	Concept	Name	Description
P	polygon type	-	Must be specified
Polygon &		polygon	the polygon to get the exterior ring from

Returns

a reference to the exterior ring

Header

Either

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

Or

```
#include <boost/geometry/core/exterior_ring.hpp>
```

exterior_ring (const version)

Function to get the exterior ring of a polygon (const version)

Synopsis

```
template<typename Polygon>
ring_return_type<Polygon const>::type exterior_ring(Polygon const & polygon)
```

Parameters

Type	Concept	Name	Description
Polygon const &	polygon type	polygon	the polygon to get the exterior ring from

Returns

a const reference to the exterior ring

Header

Either

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

Or

```
#include <boost/geometry/core/exterior_ring.hpp>
```

interior_rings

interior_rings

Function to get the interior rings of a polygon (non const version)

Synopsis

```
template<typename Polygon>
interior_return_type<Polygon>::type interior_rings(Polygon & polygon)
```

Parameters

Type	Concept	Name	Description
Polygon &	polygon type	polygon	the polygon to get the interior rings from

Returns

the interior rings (possibly a reference)

Header

Either

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

Or

```
#include <boost/geometry/core/interior_rings.hpp>
```

interior_rings (const version)

Function to get the interior rings of a polygon (const version)

Synopsis

```
template<typename Polygon>
interior_return_type<Polygon const>::type interior_rings(Polygon const & polygon)
```

Parameters

Type	Concept	Name	Description
Polygon const &	polygon type	polygon	the polygon to get the interior rings from

Returns

the interior rings (possibly a const reference)

Header

Either

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

Or

```
#include <boost/geometry/core/interior_rings.hpp>
```

Adapted models

C array

C arrays are adapted to the Boost.Geometry point concept

Description

C arrays, such as `double[2]` or `int[3]`, are (optionally) adapted to the Boost.Geometry point concept. They can therefore be used in many Boost.Geometry algorithms.

Note that a C array cannot be the point type of a linestring or a polygon. The reason for that is that a `std::vector` does not allow containing C arrays (this is not related to Boost.Geometry). The C array is therefore limited to the point type.

Model of

Point Concept

Header

```
#include <boost/geometry/geometries/adapted/c_array.hpp>
```

The standard header `boost/geometry.hpp` does not include this header.

Example

Small example showing the combination of an array with a Boost.Geometry algorithm

```
#include <iostream>
#include <boost/geometry.hpp>
#include <boost/geometry/geometries/adapted/c_array.hpp>

BOOST_GEOMETRY_REGISTER_C_ARRAY_CS(cs::cartesian)

int main()
{
    int a[3] = {1, 2, 3};
    int b[3] = {2, 3, 4};

    std::cout << boost::geometry::distance(a, b) << std::endl;

    return 0;
}
```

Output:

1.73205

Boost.Array

Boost.Array arrays are adapted to the Boost.Geometry point concept

Description

A `boost::array` is (optionally) adapted to the Boost.Geometry point concept. It can therefore be used in all Boost.Geometry algorithms.

A `boost::array` can be the point type used by the models `linestring`, `polygon`, `segment`, `box`, and `ring`

Model of

Point Concept

Header

```
#include <boost/geometry/geometries/adapted/boost_array.hpp>
```

The standard header `boost/geometry.hpp` does not include this header.

Example

Shows how to use a Boost.Array using Boost.Geometry's `distance`, `set` and `assign_values` algorithms

```
#include <iostream>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/linestring.hpp>
#include <boost/geometry/geometries/adapted/boost_array.hpp>

BOOST_GEOMETRY_REGISTER_BOOST_ARRAY_CS(cs::cartesian)

int main()
{
    boost::array<float, 2> a = { {1, 2} };
    boost::array<double, 2> b = { {2, 3} };
    std::cout << boost::geometry::distance(a, b) << std::endl;

    boost::geometry::set<0>(a, 1.1);
    boost::geometry::set<1>(a, 2.2);
    std::cout << boost::geometry::distance(a, b) << std::endl;

    boost::geometry::assign_values(b, 2.2, 3.3);
    std::cout << boost::geometry::distance(a, b) << std::endl;

    boost::geometry::model::linestring<boost::array<double, 2> > line;
    line.push_back(b);

    return 0;
}
```

Output:

1.41421
1.20416
1.55563

Boost.Fusion

Boost.Fusion adapted structs or classes are adapted to the Boost.Geometry point concept

Description

Boost.Fusion adapted structs are (optionally) adapted to the Boost.Geometry point concept. They can therefore be used in many Boost.Geometry algorithms.

Model of

Point Concept

Header

```
#include <boost/geometry/geometries/adapted/boost_fusion.hpp>
```

The standard header `boost/geometry.hpp` does not include this header.

Example

Shows how to combine Boost.Fusion with Boost.Geometry

```
#include <iostream>

#include <boost/fusion/include/adapt_struct_named.hpp>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/adapted/boost_fusion.hpp>

struct sample_point
{
    double x, y, z;
};

BOOST_FUSION_ADAPT_STRUCT(sample_point, (double, x) (double, y) (double, z))
BOOST_GEOMETRY_REGISTER_BOOST_FUSION_CS(cs::cartesian)

int main()
{
    sample_point a, b, c;

    // Set coordinates the Boost.Geometry way (one of the ways)
    boost::geometry::assign_values(a, 3, 2, 1);

    // Set coordinates the Boost.Fusion way
    boost::fusion::at_c<0>(b) = 6;
    boost::fusion::at_c<1>(b) = 5;
    boost::fusion::at_c<2>(b) = 4;

    // Set coordinates the native way
    c.x = 9;
    c.y = 8;
    c.z = 7;

    std::cout << "Distance a-b: " << boost::geometry::distance(a, b) << std::endl;
    std::cout << "Distance a-c: " << boost::geometry::distance(a, c) << std::endl;

    return 0;
}
```

Output:

```
Distance a-b: 5.19615
Distance a-c: 10.3923
```

Boost.Tuple

Boost.Tuple tuples with arithmetic elements can be used as points within Boost.Geometry

Description

Boost.Tuple fixed sized collections, such as `boost::tuple<double, double>`, are (optionally) adapted to the Boost.Geometry point concept.

Boost.Tuple pairs or triples might have mutually different types, such as a `boost::tuple<float, double>`. Boost.Geometry reports the first type as its [coordinate_type](#).

Boost.Geometry supports Boost.Tuple pairs, triples, quadruples, etc up to tuples with 10 elements (though most algorithms do not support so many dimensions).

A tuple can be the point type used by the models linestring, polygon, segment, box, and ring

Model of

Point Concept

Header

```
#include <boost/geometry/geometries/adapted/boost_tuple.hpp>
```

The standard header `boost/geometry.hpp` does not include this header.

Example

Shows how to use Boost.Tuple points in Boost.Geometry

Working with Boost.Tuples in Boost.Geometry is straightforward and shown in various other examples as well.

```
#include <iostream>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/polygon.hpp>
#include <boost/geometry/geometries/adapted/boost_tuple.hpp>

BOOST_GEOMETRY_REGISTER_BOOST_TUPLE_CS(cs::cartesian)

int main()
{
    boost::geometry::model::polygon<boost::tuple<double, double> > poly;
    poly.outer().push_back(boost::make_tuple(1.0, 2.0));
    poly.outer().push_back(boost::make_tuple(6.0, 4.0));
    poly.outer().push_back(boost::make_tuple(5.0, 1.0));
    poly.outer().push_back(boost::make_tuple(1.0, 2.0));

    std::cout << "Area: " << boost::geometry::area(poly) << std::endl;
    std::cout << "Contains (1.5, 2.5): "
        << std::boolalpha
        << boost::geometry::within(boost::make_tuple(1.5, 2.5), poly)
        << std::endl;

    return 0;
}
```

Output:

```
Area: 6.5
Contains (1.5, 2.5): false
```

Boost.Polygon

Boost.Polygon's point_data

The Boost.Polygon point type (`boost::polygon::point_data`) is adapted to the Boost.Geometry Point Concept.

Description

Boost.Polygon's points (as well as polygons) can be used by Boost.Geometry. The two libraries can therefore be used together. Using a `boost::polygon::point_data<...>`, algorithms from both Boost.Polygon and Boost.Geometry can be called.

Model of

Point Concept

Header

```
#include <boost/geometry/geometries/adapted/boost_polygon.hpp>
```

The standard header `boost/geometry.hpp` does not include this header.

Example

Shows how to use Boost.Polygon points within Boost.Geometry

```
#include <iostream>
#include <boost/geometry.hpp>
#include <boost/geometry/geometries/adapted/boost_polygon.hpp>

int main()
{
    boost::polygon::point_data<int> a(1, 2), b(3, 4);
    std::cout << "Distance (using Boost.Geometry): "
        << boost::geometry::distance(a, b) << std::endl;
    std::cout << "Distance (using Boost.Polygon): "
        << boost::polygon::euclidean_distance(a, b) << std::endl;

    return 0;
}
```

Output:

```
Distance (using Boost.Geometry): 2.82843
Distance (using Boost.Polygon): 2.82843
```

Boost.Polygon's rectangle_data

Boost.Polygon's rectangle type (`boost::polygon::rectangle_data`) is adapted to the Boost.Geometry Point Concept.

Description

Boost.Polygon's points (as well as polygons) can be used by Boost.Geometry. The two libraries can therefore be used together. Using a `boost::polygon::point_data<...>`, algorithms from both Boost.Polygon and Boost.Geometry can be called.

Model of

Box Concept

Header

```
#include <boost/geometry/geometries/adapted/boost_polygon.hpp>
```

The standard header `boost/geometry.hpp` does not include this header.

Example

Shows how to use Boost.Polygon points within Boost.Geometry

```
#include <iostream>
#include <boost/geometry.hpp>
#include <boost/geometry/geometries/adapted/boost_polygon.hpp>

int main()
{
    boost::polygon::point_data<int> a(1, 2), b(3, 4);
    std::cout << "Distance (using Boost.Geometry): "
        << boost::geometry::distance(a, b) << std::endl;
    std::cout << "Distance (using Boost.Polygon): "
        << boost::polygon::euclidean_distance(a, b) << std::endl;

    return 0;
}
```

Output:

```
Distance (using Boost.Geometry): 2.82843
Distance (using Boost.Polygon): 2.82843
```

Boost.Polygon's polygon_data

Boost.Polygon's polygon type (`boost::polygon::polygon_data`) is adapted to the Boost.Geometry Ring Concept.

Description

Boost.Polygon's points (as well as polygons) can be used by Boost.Geometry. The two libraries can therefore be used together. Using a `boost::polygon::point_data<...>`, algorithms from both Boost.Polygon and Boost.Geometry can be called.

Model of

Ring Concept

Header

```
#include <boost/geometry/geometries/adapted/boost_polygon.hpp>
```

The standard header `boost/geometry.hpp` does not include this header.

Example

Shows how to use Boost.Polygon points within Boost.Geometry

```
#include <iostream>
#include <boost/geometry.hpp>
#include <boost/geometry/geometries/adapted/boost_polygon.hpp>

int main()
{
    boost::polygon::point_data<int> a(1, 2), b(3, 4);
    std::cout << "Distance (using Boost.Geometry): "
               << boost::geometry::distance(a, b) << std::endl;
    std::cout << "Distance (using Boost.Polygon): "
               << boost::polygon::euclidean_distance(a, b) << std::endl;

    return 0;
}
```

Output:

```
Distance (using Boost.Geometry): 2.82843
Distance (using Boost.Polygon): 2.82843
```

Boost.Polygon's polygon_with_holes_data

Boost.Polygon's polygon type supporting holes (`boost::polygon::polygon_with_holes_data`) is adapted to the Boost.Geometry Polygon Concept.

Description

Boost.Polygon's points (as well as polygons) can be used by Boost.Geometry. The two libraries can therefore be used together. Using a `boost::polygon::point_data<...>`, algorithms from both Boost.Polygon and Boost.Geometry can be called.

Model of

Polygon Concept

Header

```
#include <boost/geometry/geometries/adapted/boost_polygon.hpp>
```

The standard header `boost/geometry.hpp` does not include this header.

Example

Shows how to use `Boost.Polygon` points within `Boost.Geometry`

```
#include <iostream>
#include <boost/geometry.hpp>
#include <boost/geometry/geometries/adapted/boost_polygon.hpp>

int main()
{
    boost::polygon::point_data<int> a(1, 2), b(3, 4);
    std::cout << "Distance (using Boost.Geometry): "
               << boost::geometry::distance(a, b) << std::endl;
    std::cout << "Distance (using Boost.Polygon): "
               << boost::polygon::euclidean_distance(a, b) << std::endl;

    return 0;
}
```

Output:

```
Distance (using Boost.Geometry): 2.82843
Distance (using Boost.Polygon): 2.82843
```

Boost.Range

Boost.Range filtered

`Boost.Range` filtered range adaptor is adapted to `Boost.Geometry`

Description

`Boost.Range` filtered range adaptor filters a range.

Model of

The `Boost.Range` filtered range adaptor takes over the model of the original geometry, which might be:

- a `linestring`
- a `ring`
- a `multi_point`
- a `multi_linestring`
- a `multi_polygon`

Header

```
#include <boost/geometry/geometries/adapted/boost_range/filtered.hpp>
```

The standard header `boost/geometry.hpp` does not include this header.

Example

Shows how to use a Boost.Geometry linestring, filtered by Boost.Range adaptor

```
#include <iostream>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/linestring.hpp>
#include <boost/geometry/geometries/point_xy.hpp>
#include <boost/geometry/geometries/adapted/boost_range/filtered.hpp>

struct not_two
{
    template <typename P>
    bool operator()(P const& p) const
    {
        return boost::geometry::get<1>(p) != 2;
    }
};

int main()
{
    typedef boost::geometry::model::d2::point_xy<int> xy;
    boost::geometry::model::linestring<xy> line;
    line.push_back(xy(0, 0));
    line.push_back(xy(1, 1));
    line.push_back(xy(2, 2));
    line.push_back(xy(3, 1));
    line.push_back(xy(4, 0));
    line.push_back(xy(5, 1));
    line.push_back(xy(6, 2));
    line.push_back(xy(7, 1));
    line.push_back(xy(8, 0));

    using boost::adaptors::filtered;
    std::cout
        << boost::geometry::length(line) << std::endl
        << boost::geometry::length(line | filtered(not_two())) << std::endl
        << boost::geometry::dsv(line | filtered(not_two())) << std::endl;

    return 0;
}
```

Output:

```
11.3137
9.65685
((0, 0), (1, 1), (3, 1), (4, 0), (5, 1), (7, 1), (8, 0))
```

Boost.Range reversed

Boost.Range reversed range adaptor is adapted to Boost.Geometry

Description

Boost.Range reversed range adaptor reverses a range.

Model of

The Boost.Range reversed range adaptor takes over the model of the original geometry, which might be:

- a linestring
- a ring
- a multi_point
- a multi_linestring
- a multi_polygon

Header

```
#include <boost/geometry/geometries/adapted/boost_range/reversed.hpp>
```

The standard header `boost/geometry.hpp` does not include this header.

Example

Shows how to use a Boost.Geometry linestring, reversed by Boost.Range adaptor

```
#include <iostream>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/linestring.hpp>
#include <boost/geometry/geometries/point_xy.hpp>
#include <boost/geometry/geometries/adapted/boost_range/reversed.hpp>

int main()
{
    typedef boost::geometry::model::d2::point_xy<int> xy;
    boost::geometry::model::linestring<xy> line;
    line.push_back(xy(0, 0));
    line.push_back(xy(1, 1));

    std::cout
        << boost::geometry::dsv(line | boost::adaptors::reversed)
        << std::endl;

    return 0;
}
```

Output:

```
((1, 1), (0, 0))
```

Boost.Range sliced

Boost.Range sliced range adaptor is adapted to Boost.Geometry

Description

Boost.Range sliced range adaptor creates a slice of a range (usually a linestring)

Model of

The Boost.Range sliced range adaptor takes over the model of the original geometry, which might be:

- a linestring
- a ring

- a multi_point
- a multi_linestring
- a multi_polygon

Header

```
#include <boost/geometry/geometries/adapted/boost_range/sliced.hpp>
```

The standard header `boost/geometry.hpp` does not include this header.

Example

Shows how to use a Boost.Geometry linestring, sliced by Boost.Range adaptor

```
#include <iostream>

#include <boost/assign.hpp>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/linestring.hpp>
#include <boost/geometry/geometries/point_xy.hpp>
#include <boost/geometry/geometries/adapted/boost_range/sliced.hpp>

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

    typedef boost::geometry::model::d2::point_xy<int> xy;
    boost::geometry::model::linestring<xy> line;
    line += xy(0, 0);
    line += xy(1, 1);
    line += xy(2, 2);
    line += xy(3, 3);
    line += xy(4, 4);

    std::cout
        << boost::geometry::dsv(line | boost::adaptors::sliced(1, 3)) << std::endl;

    return 0;
}
```

Output:

```
((1, 1), (2, 2))
```

Boost.Range strided

Boost.Range strided range adaptor is adapted to Boost.Geometry

Description

Boost.Range strided range adaptor makes a strided range (usually begin a linestring or ring) such that traversal is performed in steps of `n`.

Model of

The Boost.Range strided range adaptor takes over the model of the original geometry, which might be:

- a linestring
- a ring
- a multi_point
- a multi_linestring
- a multi_polygon

Header

```
#include <boost/geometry/geometries/adapted/boost_range/strided.hpp>
```

The standard header `boost/geometry.hpp` does not include this header.

Example

Shows how to use a Boost.Geometry ring, strided by Boost.Range adaptor

```
#include <iostream>

#include <boost/assign.hpp>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/point_xy.hpp>
#include <boost/geometry/geometries/ring.hpp>
#include <boost/geometry/geometries/adapted/boost_range/strided.hpp>

int main()
{
    using namespace boost::assign;
    using boost::adaptors::strided;

    typedef boost::geometry::model::d2::point_xy<int> xy;
    boost::geometry::model::ring<xy> ring;
    ring += xy(0, 0);
    ring += xy(0, 1);
    ring += xy(0, 2);
    ring += xy(1, 2);
    ring += xy(2, 2);
    ring += xy(2, 0);

    boost::geometry::correct(ring);

    std::cout
        << "Normal : " << boost::geometry::dsv(ring) << std::endl
        << "Strided: " << boost::geometry::dsv(ring | strided(2)) << std::endl;

    return 0;
}
```

Output:

```
Normal : ((0, 0), (0, 1), (0, 2), (1, 2), (2, 2), (2, 0), (0, 0))
Strided: ((0, 0), (0, 2), (2, 2), (0, 0))
```

Macro's for adaption

BOOST_GEOMETRY_REGISTER_BOX

Macro to register a box.

Description

The macro `BOOST_GEOMETRY_REGISTER_BOX` registers a box such that it is recognized by `Boost.Geometry` and that `Boost.Geometry` functionality can be used with the specified type. The box may contain template parameters, which must be specified then.

Synopsis

```
#define BOOST_GEOMETRY_REGISTER_BOX(Box, Point, MinCorner, MaxCorner)
```

Parameters

Name	Description
Box	Box type to be registered
Point	Point type on which box is based. Might be two or three-dimensional
MinCorner	minimum corner (should be public member or method)
MaxCorner	maximum corner (should be public member or method)

Header

```
#include <boost/geometry/geometries/register/box.hpp>
```

Example

Show the use of the macro `BOOST_GEOMETRY_REGISTER_BOX`

```
#include <iostream>
#include <boost/geometry.hpp>
#include <boost/geometry/geometries/register/point.hpp>
#include <boost/geometry/geometries/register/box.hpp>

struct my_point
{
    double x, y;
};

struct my_box
{
    my_point ll, ur;
};

// Register the point type
BOOST_GEOMETRY_REGISTER_POINT_2D(my_point, double, cs::cartesian, x, y)

// Register the box type, also notifying that it is based on "my_point"
BOOST_GEOMETRY_REGISTER_BOX(my_box, my_point, ll, ur)

int main()
{
    my_box b = boost::geometry::make<my_box>(0, 0, 2, 2);
    std::cout << "Area: " << boost::geometry::area(b) << std::endl;
    return 0;
}
```

Output:

```
Area: 4
```

BOOST_GEOMETRY_REGISTER_BOX_2D_4VALUES

Macro to register a box.

Description

The macro `BOOST_GEOMETRY_REGISTER_BOX_2D_4VALUES` registers a box such that it is recognized by Boost.Geometry and that Boost.Geometry functionality can be used with the specified type.

Synopsis

```
#define BOOST_GEOMETRY_REGISTER_BOX_2D_4VALUES(Box, Point, Left, Bottom, Right, Top)
```

Parameters

Name	Description
Box	Box type to be registered
Point	Point type reported as point_type by box. Must be two dimensional. Note that these box types do not contain points, but they must have a related point_type
Left	Left side (must be public member or method)
Bottom	Bottom side (must be public member or method)
Right	Right side (must be public member or method)
Top	Top side (must be public member or method)

Header

```
#include <boost/geometry/geometries/register/box.hpp>
```

Example

Show the use of the macro BOOST_GEOMETRY_REGISTER_BOX_2D_4VALUES

```
#include <iostream>
#include <boost/geometry.hpp>
#include <boost/geometry/geometries/register/point.hpp>
#include <boost/geometry/geometries/register/box.hpp>

struct my_point
{
    int x, y;
};

struct my_box
{
    int left, top, right, bottom;
};

BOOST_GEOMETRY_REGISTER_POINT_2D(my_point, int, cs::cartesian, x, y)

// Register the box type, also notifying that it is based on "my_point"
// (even if it does not contain it)
BOOST_GEOMETRY_REGISTER_BOX_2D_4VALUES(my_box, my_point, left, top, right, bottom)

int main()
{
    my_box b = boost::geometry::make<my_box>(0, 0, 2, 2);
    std::cout << "Area: " << boost::geometry::area(b) << std::endl;
    return 0;
}
```

Output:

```
Area: 4
```

BOOST_GEOMETRY_REGISTER_BOX_TEMPLATED

Macro to register a box.

Description

The macro `BOOST_GEOMETRY_REGISTER_BOX_TEMPLATED` registers a box such that it is recognized by Boost.Geometry and that Boost.Geometry functionality can be used with the specified type. The type must have one template parameter, which should be a point type, and should not be specified. Boost.Geometry takes care of inserting the template parameter. Hence all types of this templated box are registered, regardless of their point type.

Synopsis

```
#define BOOST_GEOMETRY_REGISTER_BOX_TEMPLATED(Box, MinCorner, MaxCorner)
```

Parameters

Name	Description
Box	Box type to be registered
MinCorner	minimum corner (should be public member or method)
MaxCorner	maximum corner (should be public member or method)

Header

```
#include <boost/geometry/geometries/register/box.hpp>
```

Example

Show the use of the macro `BOOST_GEOMETRY_REGISTER_BOX_TEMPLATED`

```
#include <iostream>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/point_xy.hpp>
#include <boost/geometry/geometries/register/box.hpp>

template <typename P>
struct my_box
{
    P ll, ur;
};

// Register the box type
BOOST_GEOMETRY_REGISTER_BOX_TEMPLATED(my_box, ll, ur)

int main()
{
    typedef my_box<boost::geometry::model::d2::point_xy<double> > box;
    box b = boost::geometry::make<box>(0, 0, 2, 2);
    std::cout << "Area: " << boost::geometry::area(b) << std::endl;
    return 0;
}
```

Output:

Area: 4

BOOST_GEOMETRY_REGISTER_LINESTRING

Macro to register a linestring.

Description

The macro BOOST_GEOMETRY_REGISTER_LINESTRING registers a linestring such that it is recognized by Boost.Geometry and that Boost.Geometry functionality can be used with the specified type. The linestring may contain template parameters, which must be specified then.

Synopsis

```
#define BOOST_GEOMETRY_REGISTER_LINESTRING(Linestring)
```

Parameters

Name	Description
Linestring	linestring type to be registered

Header

```
#include <boost/geometry/geometries/register/linestring.hpp>
```

Example

Show the use of BOOST_GEOMETRY_REGISTER_LINESTRING

```
#include <iostream>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/point_xy.hpp>
#include <boost/geometry/geometries/register/linestring.hpp>

typedef boost::geometry::model::d2::point_xy<double> point_2d;

BOOST_GEOMETRY_REGISTER_LINESTRING(std::vector<point_2d>)

int main()
{
    // Normal usage of std::
    std::vector<point_2d> line;
    line.push_back(point_2d(1, 1));
    line.push_back(point_2d(2, 2));
    line.push_back(point_2d(3, 1));

    // Usage of Boost.Geometry's length and wkt functions
    std::cout << "Length: "
        << boost::geometry::length(line)
        << std::endl;

    std::cout << "WKT: "
        << boost::geometry::wkt(line)
        << std::endl;

    return 0;
}
```

Output:

```
Length: 2.82843
WKT: LINESTRING(1 1,2 2,3 1)
```

BOOST_GEOMETRY_REGISTER_LINESTRING_TEMPLATED

Macro to register a templated linestring.

Description

The macro `BOOST_GEOMETRY_REGISTER_LINESTRING_TEMPLATED` registers a templated linestring such that it is recognized by Boost.Geometry and that Boost.Geometry functionality can be used with the specified type. The type must have one template parameter, which should be a point type, and should not be specified. Boost.Geometry takes care of inserting the template parameter. Hence all types of this templated linestring are registered, regardless of their point type.

Synopsis

```
#define BOOST_GEOMETRY_REGISTER_LINESTRING_TEMPLATED(Linestring)
```

Parameters

Name	Description
Linestring	linestring (without template parameters) type to be registered

Header

```
#include <boost/geometry/geometries/register/linestring.hpp>
```

Example

Show the use of the macro `BOOST_GEOMETRY_REGISTER_LINESTRING_TEMPLATED`

```
#include <iostream>
#include <deque>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/point_xy.hpp>
#include <boost/geometry/geometries/register/linestring.hpp>

// Adapt any deque to Boost.Geometry Linestring Concept
BOOST_GEOMETRY_REGISTER_LINESTRING_TEMPLATED(std::deque)

int main()
{
    std::deque<boost::geometry::model::d2::point_xy<double> > line(2);
    boost::geometry::assign_values(line[0], 1, 1);
    boost::geometry::assign_values(line[1], 2, 2);

    // Boost.Geometry algorithms work on any deque now
    std::cout << "Length: " << boost::geometry::length(line) << std::endl;
    std::cout << "Line: " << boost::geometry::dsv(line) << std::endl;

    return 0;
}
```

Output:


```
Length: 1.41421
Line: ((1, 1), (2, 2))
```

BOOST_GEOMETRY_REGISTER_MULTI_LINESTRING

Macro to register a multi_linestring.

Description

The macro BOOST_GEOMETRY_REGISTER_MULTI_LINESTRING registers a multi_linestring such that it is recognized by Boost.Geometry and that Boost.Geometry functionality can be used with the specified type. The multi_linestring may contain template parameters, which must be specified then.

Synopsis

```
#define BOOST_GEOMETRY_REGISTER_MULTI_LINESTRING(MultiLineString)
```

Parameters

Name	Description
MultiLineString	multi_linestring type to be registered

Header

```
#include <boost/geometry/multi/geometries/register/multi_linestring.hpp>
```

Example

Show the use of the macro BOOST_GEOMETRY_REGISTER_MULTI_LINESTRING

```
#include <iostream>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/linestring.hpp>
#include <boost/geometry/geometries/adapted/boost_tuple.hpp>
#include <boost/geometry/multi/geometries/register/multi_linestring.hpp>

typedef boost::geometry::model::linestring
<
    boost::tuple<float, float>
> linestring_type;

BOOST_GEOMETRY_REGISTER_BOOST_TUPLE_CS(cs::cartesian)
BOOST_GEOMETRY_REGISTER_MULTI_LINESTRING(std::deque<linestring_type>)

int main()
{
    // Normal usage of std::
    std::deque<linestring_type> lines(2);
    boost::geometry::read_wkt("LINESTRING(0 0,1 1)", lines[0]);
    boost::geometry::read_wkt("LINESTRING(2 2,3 3)", lines[1]);

    // Usage of Boost.Geometry
    std::cout << "LENGTH: " << boost::geometry::length(lines) << std::endl;

    return 0;
}
```

Output:

```
LENGTH: 2.82843
```

BOOST_GEOMETRY_REGISTER_MULTI_LINESTRING_TEMPLATED

Macro to register a templated multi_linestring.

Description

The macro `BOOST_GEOMETRY_REGISTER_MULTI_LINESTRING_TEMPLATED` registers a templated multi_linestring such that it is recognized by Boost.Geometry and that Boost.Geometry functionality can be used with the specified type. The type must have one template parameter, which should be a linestring type, and should not be specified. Boost.Geometry takes care of inserting the template parameter. Hence all types of this templated multi_linestring are registered, regardless of their point type.

Synopsis

```
#define BOOST_GEOMETRY_REGISTER_MULTI_LINESTRING_TEMPLATED(MultiLineString)
```

Parameters

Name	Description
MultiLineString	multi_linestring (without template parameters) type to be registered

Header

```
#include <boost/geometry/multi/geometries/register/multi_linestring.hpp>
```

Example

```
[register_multi_linestring_templated] [register_multi_linestring_templated_output]
```

BOOST_GEOMETRY_REGISTER_MULTI_POINT

Macro to register a multi_point.

Description

The macro `BOOST_GEOMETRY_REGISTER_MULTI_POINT` registers a multi_point such that it is recognized by Boost.Geometry and that Boost.Geometry functionality can be used with the specified type. The multi_point may contain template parameters, which must be specified then.

Synopsis

```
#define BOOST_GEOMETRY_REGISTER_MULTI_POINT(MultiPoint)
```

Parameters

Name	Description
MultiPoint	multi_point type to be registered

Header

```
#include <boost/geometry/multi/geometries/register/multi_point.hpp>
```

Example

Show the use of the macro `BOOST_GEOMETRY_REGISTER_MULTI_POINT`

```
#include <iostream>
#include <boost/geometry.hpp>
#include <boost/geometry/geometries/adapted/boost_tuple.hpp>
#include <boost/geometry/multi/geometries/register/multi_point.hpp>
#include <boost/geometry/multi/io/wkt/wkt.hpp>

typedef boost::tuple<float, float> point_type;

BOOST_GEOMETRY_REGISTER_BOOST_TUPLE_CS(cs::cartesian)
BOOST_GEOMETRY_REGISTER_MULTI_POINT(std::deque< point_type >)

int main()
{
    // Normal usage of std::
    std::deque<point_type> multi_point;
    multi_point.push_back(point_type(1, 1));
    multi_point.push_back(point_type(3, 2));

    // Usage of Boost.Geometry
    std::cout << "WKT: " << boost::geometry::wkt(multi_point) << std::endl;

    return 0;
}
```

Output:

```
WKT: MULTIPOINT((1 1),(3 2))
```

BOOST_GEOMETRY_REGISTER_MULTI_POINT_TEMPLATED

Macro to register a templated `multi_point`.

Description

The macro `BOOST_GEOMETRY_REGISTER_MULTI_POINT_TEMPLATED` registers a templated `multi_point` such that it is recognized by `Boost.Geometry` and that `Boost.Geometry` functionality can be used with the specified type. The type must have one template parameter, which should be a point type, and should not be specified. `Boost.Geometry` takes care of inserting the template parameter. Hence all types of this templated `multi_point` are registered, regardless of their point type.

Synopsis

```
#define BOOST_GEOMETRY_REGISTER_MULTI_POINT_TEMPLATED(MultiPoint)
```

Parameters

Name	Description
MultiPoint	<code>multi_point</code> (without template parameters) type to be registered

Header

```
#include <boost/geometry/multi/geometries/register/multi_point.hpp>
```

Example

Show the use of the macro `BOOST_GEOMETRY_REGISTER_MULTI_POINT_TEMPLATED`

```
#include <iostream>
#include <boost/geometry.hpp>
#include <boost/geometry/geometries/adapted/boost_tuple.hpp>
#include <boost/geometry/multi/geometries/register/multi_point.hpp>
#include <boost/geometry/multi/io/wkt/wkt.hpp>

BOOST_GEOMETRY_REGISTER_MULTI_POINT_TEMPLATED(std::deque)

BOOST_GEOMETRY_REGISTER_BOOST_TUPLE_CS(cs::cartesian)

int main()
{
    // Normal usage of std::
    std::deque<boost::tuple<float, float> > multi_point;
    multi_point.push_back(boost::tuple<float, float>(1, 1));
    multi_point.push_back(boost::tuple<float, float>(3, 2));

    // Usage of Boost.Geometry
    std::cout << "WKT: " << boost::geometry::wkt(multi_point) << std::endl;

    return 0;
}
```

Output:

```
WKT: MULTIPOINT((1 1),(3 2))
```

BOOST_GEOMETRY_REGISTER_MULTI_POLYGON

Macro to register a multi_polygon.

Description

The macro `BOOST_GEOMETRY_REGISTER_MULTI_POLYGON` registers a multi_polygon such that it is recognized by Boost.Geometry and that Boost.Geometry functionality can be used with the specified type. The multi_polygon may contain template parameters, which must be specified then.

Synopsis

```
#define BOOST_GEOMETRY_REGISTER_MULTI_POLYGON(MultiPolygon)
```

Parameters

Name	Description
MultiPolygon	multi_polygon type to be registered

Header

```
#include <boost/geometry/multi/geometries/register/multi_polygon.hpp>
```

Example

Show the use of the macro `BOOST_GEOMETRY_REGISTER_MULTI_POLYGON`

```
#include <iostream>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/polygon.hpp>
#include <boost/geometry/geometries/adapted/boost_tuple.hpp>
#include <boost/geometry/multi/geometries/register/multi_polygon.hpp>

typedef boost::geometry::model::polygon
<
    boost::tuple<float, float>
> polygon_type;

BOOST_GEOMETRY_REGISTER_BOOST_TUPLE_CS(cs::cartesian)
BOOST_GEOMETRY_REGISTER_MULTI_POLYGON(std::vector<polygon_type>)

int main()
{
    // Normal usage of std::
    std::vector<polygon_type> polygons(2);
    boost::geometry::read_wkt("POLYGON((0 0,0 1,1 1,1 0,0 0))", polygons[0]);
    boost::geometry::read_wkt("POLYGON((3 0,3 1,4 1,4 0,3 0))", polygons[1]);

    // Usage of Boost.Geometry
    std::cout << "AREA: " << boost::geometry::area(polygons) << std::endl;

    return 0;
}
```

Output:

```
AREA: 2
```

BOOST_GEOMETRY_REGISTER_MULTI_POLYGON_TEMPLATED

Macro to register a templated multi_polygon.

Description

The macro `BOOST_GEOMETRY_REGISTER_MULTI_POLYGON_TEMPLATED` registers a templated multi_polygon such that it is recognized by Boost.Geometry and that Boost.Geometry functionality can be used with the specified type. The type must have one template parameter, which should be a polygon type, and should not be specified. Boost.Geometry takes care of inserting the template parameter. Hence all types of this templated multi_polygon are registered, regardless of their point type.

Synopsis

```
#define BOOST_GEOMETRY_REGISTER_MULTI_POLYGON_TEMPLATED(MultiPolygon)
```

Parameters

Name	Description
MultiPolygon	multi_polygon (without template parameters) type to be registered

Header

```
#include <boost/geometry/multi/geometries/register/multi_polygon.hpp>
```

Example

```
[register_multi_polygon_templated] [register_multi_polygon_templated_output]
```

BOOST_GEOMETRY_REGISTER_POINT_2D

Macro to register a 2D point type.

Description

The macro `BOOST_GEOMETRY_REGISTER_POINT_2D` registers a two-dimensional point type such that it is recognized by `Boost.Geometry` and that `Boost.Geometry` functionality can be used with the specified type.

Synopsis

```
#define BOOST_GEOMETRY_REGISTER_POINT_2D(Point, CoordinateType, CoordinateSystem, Field0, Field1)
```

Parameters

Name	Description
Point	Point type to be registered
CoordinateType	Type of the coordinates of the point (e.g. double)
CoordinateSystem	Coordinate system (e.g. <code>cs::cartesian</code>)
Field0	Member containing first (usually x) coordinate
Field1	Member containing second (usually y) coordinate

Header

```
#include <boost/geometry/geometries/register/point.hpp>
```



Caution

Use the macro outside any namespace



Note

A point can include a namespace

Examples

Show the use of the macro `BOOST_GEOMETRY_REGISTER_POINT_2D`

```

#include <iostream>
#include <boost/geometry.hpp>
#include <boost/geometry/geometries/register/point.hpp>

❶
struct legacy_point
{
    double x, y;
};

BOOST_GEOMETRY_REGISTER_POINT_2D(legacy_point, double, cs::cartesian, x, y) ❷

int main()
{
    legacy_point p1, p2;

    namespace bg = boost::geometry;

    ❸
    bg::assign_values(p1, 1, 1);
    bg::assign_values(p2, 2, 2);

    double d = bg::distance(p1, p2);

    std::cout << "Distance: " << d << std::endl;

    return 0;
}

```

- ❶ Somewhere, any legacy point struct is defined
- ❷ The magic: adapt it to Boost.Geometry Point Concept
- ❸ Any Boost.Geometry function can be used for legacy point now. Here: assign_values and distance

Output:

```
Distance: 1.41421
```

BOOST_GEOMETRY_REGISTER_POINT_2D_CONST

Macro to register a 2D point type (const version)

Description

The macro `BOOST_GEOMETRY_REGISTER_POINT_2D_CONST` registers a two-dimensional point type such that it is recognized by Boost.Geometry and that Boost.Geometry functionality can be used with the specified type. The const version registers only read access to the fields, the point type is therefore read-only

Synopsis

```

#define BOOST_GEOMETRY_REGISTER_POINT_2D_CONST(Point, CoordinateType, CoordinateSystem, Field0, Field1)

```

Parameters

Name	Description
Point	Point type to be registered
CoordinateType	Type of the coordinates of the point (e.g. double)
CoordinateSystem	Coordinate system (e.g. cs::cartesian)
Field0	Member containing first (usually x) coordinate
Field1	Member containing second (usually y) coordinate

Header

```
#include <boost/geometry/geometries/register/point.hpp>
```

BOOST_GEOMETRY_REGISTER_POINT_2D_GET_SET

Macro to register a 2D point type (having separate get/set methods)

Description

The macro `BOOST_GEOMETRY_REGISTER_POINT_2D_GET_SET` registers a two-dimensional point type such that it is recognized by Boost.Geometry and that Boost.Geometry functionality can be used with the specified type.. The get/set version registers get and set methods separately and can be used for classes with protected member variables and get/set methods to change coordinates

Synopsis

```
#define BOOST_GEOMETRY_REGISTER_POINT_2D_GET_SET(Point, CoordinateType, CoordinateSystem, Get0, Get1, Set0, Set1)
```

Parameters

Name	Description
Point	Point type to be registered
CoordinateType	Type of the coordinates of the point (e.g. double)
CoordinateSystem	Coordinate system (e.g. cs::cartesian)
Get0	Method to get the first (usually x) coordinate
Get1	Method to get the second (usually y) coordinate
Set0	Method to set the first (usually x) coordinate
Set1	Method to set the second (usually y) coordinate

Header

```
#include <boost/geometry/geometries/register/point.hpp>
```


BOOST_GEOMETRY_REGISTER_POINT_3D

Macro to register a 3D point type.

Description

The macro `BOOST_GEOMETRY_REGISTER_POINT_3D` registers a three-dimensional point type such that it is recognized by `Boost.Geometry` and that `Boost.Geometry` functionality can be used with the specified type.

Synopsis

```
#define BOOST_GEOMETRY_REGISTER_POINT_3D(Point, CoordinateType, CoordinateSystem, Field0, Field1, Field2)
```

Parameters

Name	Description
Point	Point type to be registered
CoordinateType	Type of the coordinates of the point (e.g. double)
CoordinateSystem	Coordinate system (e.g. <code>cs::cartesian</code>)
Field0	Member containing first (usually x) coordinate
Field1	Member containing second (usually y) coordinate
Field2	Member containing third (usually z) coordinate

Header

```
#include <boost/geometry/geometries/register/point.hpp>
```

BOOST_GEOMETRY_REGISTER_POINT_3D_CONST

Macro to register a 3D point type (const version)

Description

The macro `BOOST_GEOMETRY_REGISTER_POINT_3D_CONST` registers a three-dimensional point type such that it is recognized by `Boost.Geometry` and that `Boost.Geometry` functionality can be used with the specified type.. The const version registers only read access to the fields, the point type is therefore read-only

Synopsis

```
#define BOOST_GEOMETRY_REGISTER_POINT_3D_CONST(Point, CoordinateType, CoordinateSystem, Field0, Field1, Field2)
```

Parameters

Name	Description
Point	Point type to be registered
CoordinateType	Type of the coordinates of the point (e.g. double)
CoordinateSystem	Coordinate system (e.g. cs::cartesian)
Field0	Member containing first (usually x) coordinate
Field1	Member containing second (usually y) coordinate
Field2	Member containing third (usually z) coordinate

Header

```
#include <boost/geometry/geometries/register/point.hpp>
```

BOOST_GEOMETRY_REGISTER_POINT_3D_GET_SET

Macro to register a 3D point type (having separate get/set methods)

Description

The macro `BOOST_GEOMETRY_REGISTER_POINT_3D_GET_SET` registers a three-dimensional point type such that it is recognized by Boost.Geometry and that Boost.Geometry functionality can be used with the specified type. The get/set version registers get and set methods separately and can be used for classes with protected member variables and get/set methods to change coordinates.

Synopsis

```
#define BOOST_GEOMETRY_REGISTER_POINT_3D_GET_SET(Point, CoordinateType, CoordinateSystem, Get0, Get1, Get2, Set0, Set1, Set2)
```

Parameters

Name	Description
Point	Point type to be registered
CoordinateType	Type of the coordinates of the point (e.g. double)
CoordinateSystem	Coordinate system (e.g. cs::cartesian)
Get0	Method to get the first (usually x) coordinate
Get1	Method to get the second (usually y) coordinate
Get2	Method to get the third (usually z) coordinate
Set0	Method to set the first (usually x) coordinate
Set1	Method to set the second (usually y) coordinate
Set2	Method to set the third (usually z) coordinate

Header

```
#include <boost/geometry/geometries/register/point.hpp>
```

BOOST_GEOMETRY_REGISTER_RING

Macro to register a ring.

Description

The macro `BOOST_GEOMETRY_REGISTER_RING` registers a ring such that it is recognized by `Boost.Geometry` and that `Boost.Geometry` functionality can be used with the specified type. The ring may contain template parameters, which must be specified then.

Synopsis

```
#define BOOST_GEOMETRY_REGISTER_RING(Ring)
```

Parameters

Name	Description
Ring	ring type to be registered

Header

```
#include <boost/geometry/geometries/register/ring.hpp>
```

Example

Show the use of the macro `BOOST_GEOMETRY_REGISTER_RING`

```
#include <iostream>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/point_xy.hpp>
#include <boost/geometry/geometries/register/ring.hpp>

typedef boost::geometry::model::d2::point_xy<double> point_2d;

BOOST_GEOMETRY_REGISTER_RING(std::vector<point_2d>) ❶

int main()
{
    // Normal usage of std::
    std::vector<point_2d> ring;
    ring.push_back(point_2d(1, 1));
    ring.push_back(point_2d(2, 2));
    ring.push_back(point_2d(2, 1));

    // Usage of Boost.Geometry
    boost::geometry::correct(ring);
    std::cout << "Area: " << boost::geometry::area(ring) << std::endl;
    std::cout << "WKT: " << boost::geometry::wkt(ring) << std::endl;

    return 0;
}
```

❶ The magic: adapt vector to Boost.Geometry Ring Concept

Output:

```
Area: 0.5  
WKT: POLYGON((1 1,2 2,2 1,1 1))
```

BOOST_GEOMETRY_REGISTER_RING_TEMPLATED

Macro to register a templated ring.

Description

The macro `BOOST_GEOMETRY_REGISTER_RING_TEMPLATED` registers a templated ring such that it is recognized by Boost.Geometry and that Boost.Geometry functionality can be used with the specified type. The type must have one template parameter, which should be a point type, and should not be specified. Boost.Geometry takes care of inserting the template parameter. Hence all types of this templated ring are registered, regardless of their point type.

Synopsis

```
#define BOOST_GEOMETRY_REGISTER_RING_TEMPLATED(Ring)
```

Parameters

Name	Description
Ring	ring (without template parameters) type to be registered

Header

```
#include <boost/geometry/geometries/register/ring.hpp>
```

ExampleShow the use of the macro `BOOST_GEOMETRY_REGISTER_RING_TEMPLATED`

```
#include <iostream>
#include <deque>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/point_xy.hpp>
#include <boost/geometry/geometries/register/ring.hpp>

// Adapt any deque to Boost.Geometry Ring Concept
BOOST_GEOMETRY_REGISTER_RING_TEMPLATED(std::deque)

int main()
{
    std::deque<boost::geometry::model::d2::point_xy<double> > ring(3);
    boost::geometry::assign_values(ring[0], 0, 0);
    boost::geometry::assign_values(ring[2], 4, 1);
    boost::geometry::assign_values(ring[1], 1, 4);

    // Boost.Geometry algorithms work on any deque now
    boost::geometry::correct(ring);
    std::cout << "Area: " << boost::geometry::area(ring) << std::endl;
    std::cout << "Contents: " << boost::geometry::wkt(ring) << std::endl;

    return 0;
}
```

Output:

```
Area: 7.5
Line: ((0, 0), (1, 4), (4, 1), (0, 0))
```

Algorithms

area

area

Calculates the area of a geometry.

Description

The free function `area` calculates the area of a geometry. It uses the default strategy, based on the coordinate system of the geometry.

The area algorithm calculates the surface area of all geometries having a surface, namely box, polygon, ring, multipolygon. The units are the square of the units used for the points defining the surface. If subject geometry is defined in meters, then area is calculated in square meters.

The area calculation can be done in all three common coordinate systems, Cartesian, Spherical and Geographic as well.

Synopsis

```
template<typename Geometry>
default_area_result<Geometry>::type area(Geometry const & geometry)
```

Parameters

Type	Concept	Name	Description
Geometry const &	Any type fulfilling a Geometry Concept	geometry	A model of the specified concept

Returns

The calculated area

Header

Either

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

Or

```
#include <boost/geometry/algorithms/area.hpp>
```










Conformance

The function area implements function Area from the [OGC Simple Feature Specification](#).

Behavior

Case	Behavior
pointlike (e.g. point)	Returns 0
linear (e.g. linestring)	Returns 0
areal (e.g. polygon)	Returns the area
Cartesian	Returns the area in the same units as the input coordinates
Spherical	Returns the area on a unit sphere (or another sphere, if specified as such in the constructor of the strategy)
Reversed polygon (coordinates not according their orientation)	Returns the negative area

Supported geometries

Geometry	Status
Point	
Segment	
Box	
Linestring	
Ring	
Polygon	
MultiPoint	
MultiLinestring	
MultiPolygon	

Complexity

Linear

Examples

Calculate the area of a polygon

```
#include <iostream>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/point_xy.hpp>
#include <boost/geometry/geometries/polygon.hpp>
#include <boost/geometry/io/wkt/wkt.hpp>

namespace bg = boost::geometry; ❶

int main()
{
    // Calculate the area of a cartesian polygon
    bg::model::polygon<bg::model::d2::point_xy<double> > poly;
    bg::read_wkt("POLYGON((0 0,0 7,4 2,2 0,0 0))", poly);
    double area = bg::area(poly);
    std::cout << "Area: " << area << std::endl;

    // Calculate the area of a spherical equatorial polygon
    bg::model::polygon<bg::model::point<float, 2, bg::cs::spherical_equatorial<bg::degree> > > sph_poly;
    bg::read_wkt("POLYGON((0 0,0 45,45 0,0 0))", sph_poly);
    area = bg::area(sph_poly);
    std::cout << "Area: " << area << std::endl;

    return 0;
}
```

❶ Convenient namespace alias

Output:

```
Area: 16
Area: 0.339837
```

area (with strategy)

Calculates the area of a geometry using the specified strategy.

Description

The free function `area` calculates the area of a geometry using the specified strategy. Reasons to specify a strategy include: use another coordinate system for calculations; construct the strategy beforehand (e.g. with the radius of the Earth); select a strategy when there are more than one available for a calculation.

Synopsis

```
template<typename Geometry, typename Strategy>
Strategy::return_type area(Geometry const & geometry, Strategy const & strategy)
```

Parameters

Type	Concept	Name	Description
Geometry const &	Any type fulfilling a Geometry Concept	geometry	A model of the specified concept
Strategy const &	Any type fulfilling a Area Strategy Concept	strategy	The strategy which will be used for area calculations

Returns

The calculated area

Header

Either

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

Or

```
#include <boost/geometry/algorithms/area.hpp>
```










Conformance

The function `area` implements function `Area` from the [OGC Simple Feature Specification](#).

Behavior

Case	Behavior
pointlike (e.g. point)	Returns 0
linear (e.g. linestring)	Returns 0
areal (e.g. polygon)	Returns the area
Cartesian	Returns the area in the same units as the input coordinates
Spherical	Returns the area on a unit sphere (or another sphere, if specified as such in the constructor of the strategy)
Reversed polygon (coordinates not according their orientation)	Returns the negative area

Supported geometries

Geometry	Status
Point	
Segment	
Box	
Linestring	
Ring	
Polygon	
MultiPoint	
MultiLinestring	
MultiPolygon	

Complexity

Linear

Example

Calculate the area of a polygon

```

#include <iostream>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/point_xy.hpp>
#include <boost/geometry/geometries/polygon.hpp>
#include <boost/geometry/io/wkt/wkt.hpp>

namespace bg = boost::geometry; ❶

int main()
{
    // Calculate the area of a cartesian polygon
    bg::model::polygon<bg::model::d2::point_xy<double> > poly;
    bg::read_wkt("POLYGON((0 0,0 7,4 2,2 0,0 0))", poly);
    double area = bg::area(poly);
    std::cout << "Area: " << area << std::endl;

    // Calculate the area of a spherical polygon (for latitude: 0 at equator)
    bg::model::polygon<bg::model::point<float, 2, bg::cs::spherical_equatorial<bg::deJ
gree> > > sph_poly;
    bg::read_wkt("POLYGON((0 0,0 45,45 0,0 0))", sph_poly);
    area = bg::area(sph_poly);
    std::cout << "Area: " << area << std::endl;

    return 0;
}

```

❶ Convenient namespace alias

Output:

```

Area: 16
Area: 0.339837

```

Available Strategies

- Surveyor (cartesian)
- Huiller (spherical)

assign

assign

Assigns one geometry to another geometry.

Description

The assign algorithm assigns one geometry, e.g. a BOX, to another geometry, e.g. a RING. This only if it is possible and applicable.

Synopsis

```

template<typename Geometry1, typename Geometry2>
void assign(Geometry1 & geometry1, Geometry2 const & geometry2)

```

Parameters

Type	Concept	Name	Description
Geometry1 &	Any type fulfilling a Geometry Concept	geometry1	A model of the specified concept (target)
Geometry2 const &	Any type fulfilling a Geometry Concept	geometry2	A model of the specified concept (source)

Header

Either

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

Or

```
#include <boost/geometry/algorithms/assign.hpp>
```

Example

Shows how to assign a geometry from another geometry

```

#include <iostream>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/box.hpp>
#include <boost/geometry/geometries/point_xy.hpp>
#include <boost/geometry/geometries/polygon.hpp>
#include <boost/geometry/geometries/adapted/boost_tuple.hpp>

BOOST_GEOMETRY_REGISTER_BOOST_TUPLE_CS(cs::cartesian)

int main()
{
    typedef boost::geometry::model::d2::point_xy<double> point;
    typedef boost::geometry::model::box<point> box;
    typedef boost::geometry::model::polygon<point> polygon;

    point p1;
    box b;
    boost::geometry::assign_values(p1, 1, 1);
    boost::geometry::assign_values(b, 1, 1, 2, 2);

    // Assign a box to a polygon (target = source)
    polygon p;
    boost::geometry::assign(p, b);

    // Assign a point to another point type (conversion of point-type)
    boost::tuple<double, double> p2;
    boost::geometry::assign(p2, p1);

    using boost::geometry::dsv;
    std::cout
        << "box: " << dsv(b) << std::endl
        << "polygon: " << dsv(p) << std::endl
        << "point: " << dsv(p1) << std::endl
        << "point tuples: " << dsv(p2) << std::endl
        ;

    return 0;
}

```

Output:

```

box: ((1, 1), (2, 2))
polygon: (((1, 1), (1, 2), (2, 2), (2, 1), (1, 1)))
point: (1, 1)
point tuples: (1, 1)

```

See also

- [convert](#)

assign_inverse

assign to a box inverse infinite

Description

The `assign_inverse` function initialize a 2D or 3D box with large coordinates, the min corner is very large, the max corner is very small. This is a convenient starting point to collect the minimum bounding box of a geometry.

Synopsis

```
template<typename Geometry>
void assign_inverse(Geometry & geometry)
```

Parameters

Type	Concept	Name	Description
Geometry &	Any type fulfilling a Geometry Concept	geometry	A model of the specified concept

Header

Either

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

Or

```
#include <boost/geometry/algorithms/assign.hpp>
```

Example

Usage of `assign_inverse` and `expand` to conveniently determine bounding 3D box of two points

```
#include <iostream>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/box.hpp>
#include <boost/geometry/geometries/point.hpp>

using namespace boost::geometry;

int main()
{
    typedef model::point<float, 3, cs::cartesian> point;
    typedef model::box<point> box;

    box all;
    assign_inverse(all);
    std::cout << dsv(all) << std::endl;
    expand(all, point(0, 0, 0));
    expand(all, point(1, 2, 3));
    std::cout << dsv(all) << std::endl;

    return 0;
}
```

Output:

```
((3.40282e+038, 3.40282e+038, 3.40282e+038), (-3.40282e+038, -3.40282e+038, -3.40282e+038))
((0, 0, 0), (1, 2, 3))
```

See also

- [make_inverse](#)

assign_points

Assign a range of points to a linestring, ring or polygon.

Synopsis

```
template<typename Geometry, typename Range>
void assign_points(Geometry & geometry, Range const & range)
```

Parameters

Type	Concept	Name	Description
Geometry &	Any type fulfilling a Geometry Concept	geometry	A model of the specified concept
Range const &	Any type fulfilling a Range Concept where its range_value type fulfills the Point Concept	range	A range containing points fulfilling range and point concepts

Header

Either

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

Or

```
#include <boost/geometry/algorithms/assign.hpp>
```

Notes



Note

Assign automatically clears the geometry before assigning (use append if you don't want that)

Example

Shows usage of Boost.Geometry's assign, Boost.Assign, and Boost.Range to assign ranges of a linestring

```

#include <iostream>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/linestring.hpp>
#include <boost/geometry/geometries/adapted/boost_tuple.hpp>

#include <boost/assign.hpp>
#include <boost/geometry/geometries/adapted/boost_range/filtered.hpp>

BOOST_GEOMETRY_REGISTER_BOOST_TUPLE_CS(cs::cartesian)

template <typename T>
struct x_between
{
    x_between(T a, T b)
        : fa(a), fb(b)
    {}

    template <typename P>
    bool operator()(P const& p) const
    {
        return boost::geometry::get<0>(p) >= fa
            && boost::geometry::get<0>(p) <= fb;
    }
private :
    T fa, fb;
};

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

    typedef boost::geometry::model::linestring<boost::tuple<int, int> > ls;

    ls line1, line2, line3;

    line1 = tuple_list_of(0, 0)(2, 3)(4, 0)(6, 3)(8, 0)(10, 3)(12, 0); ❶
    boost::geometry::assign_points(line2, tuple_list_of(0, 0)(2, 2)(4, 0)(6, 2)(8, 0)); ❷
    boost::geometry::assign_points(line3, line1 | boost::adap
    tortors::filtered(x_between<int>(4, 8))); ❸

    std::cout << "line 1: " << boost::geometry::dsv(line1) << std::endl;
    std::cout << "line 2: " << boost::geometry::dsv(line2) << std::endl;
    std::cout << "line 3: " << boost::geometry::dsv(line3) << std::endl;

    return 0;
}

```

- ❶ tuple_list_of is part of Boost.Assign and can be used for Boost.Geometry if points are tuples
- ❷ tuple_list_of delivers a range and can therefore be used in boost::geometry::assign
- ❸ Boost.Range adaptors can also be used in boost::geometry::assign

Output:

```

line 1: ((0, 0), (2, 3), (4, 0), (6, 3), (8, 0), (10, 3), (12, 0))
line 2: ((0, 0), (2, 2), (4, 0), (6, 2), (8, 0))
line 3: ((4, 0), (6, 3), (8, 0))

```

See also

- [append](#)

assign_values (2 coordinate values)

Assign two coordinates to a geometry (usually a 2D point)

Synopsis

```
template<typename Geometry, typename Type>
void assign_values(Geometry & geometry, Type const & c1, Type const & c2)
```

Parameters

Type	Concept	Name	Description
Geometry &	Any type fulfilling a Geometry Concept	geometry	A model of the specified concept
Type const &	numerical type (int, double, ttmath, ...) to specify the co-ordinates	c1	First coordinate (usually x-coordinate)
Type const &	numerical type (int, double, ttmath, ...) to specify the co-ordinates	c2	Second coordinate (usually y-coordinate)

Header

```
#include <boost/geometry/algorithms/detail/assign_values.hpp>
```

Example

Shows the usage of assign to set point coordinates, and, besides that, shows how you can initialize ttmath points with high precision


```

#include <iostream>
#include <iomanip>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/point_xy.hpp>

#if defined(HAVE_TTMATH)
# include <boost/geometry/extensions/contrib/ttmath_stub.hpp>
#endif

int main()
{
    using boost::geometry::assign_values;

    boost::geometry::model::d2::point_xy<double> p1;
    assign_values(p1, 1.2345, 2.3456);

    #if defined(HAVE_TTMATH)
    boost::geometry::model::d2::point_xy<ttmath::Big<1,4> > p2;
    assign_values(p2, "1.2345", "2.3456"); ❶
    #endif

    std::cout
        << std::setprecision(20)
        << boost::geometry::dsv(p1) << std::endl
    #if defined(HAVE_TTMATH)
        << boost::geometry::dsv(p2) << std::endl
    #endif
        ;

    return 0;
}

```

- ❶ It is possible to assign coordinates with other types than the coordinate type. For ttmath, you can e.g. conveniently use strings. The advantage is that it then has higher precision, because if doubles are used for assignments the double-precision is used.

Output:

```

(1.2344999999999999, 2.34560000000000001)
(1.2345, 2.3456)

```

See also

- [make](#)

assign_values (4 coordinate values)

Assign four values to a geometry (usually a box or segment)

Synopsis

```

template<typename Geometry, typename Type>
void assign_values(Geometry & geo,
    metry, Type const & c1, Type const & c2, Type const & c3, Type const & c4)

```

Parameters

Type	Concept	Name	Description
Geometry &	Any type fulfilling a Geometry Concept	geometry	A model of the specified concept
Type const &	numerical type (int, double, ttmath, ...) to specify the co-ordinates	c1	First coordinate (usually x1)
Type const &	numerical type (int, double, ttmath, ...) to specify the co-ordinates	c2	Second coordinate (usually y1)
Type const &	numerical type (int, double, ttmath, ...) to specify the co-ordinates	c3	Third coordinate (usually x2)
Type const &	numerical type (int, double, ttmath, ...) to specify the co-ordinates	c4	Fourth coordinate (usually y2)

Header

```
#include <boost/geometry/algorithms/detail/assign_values.hpp>
```

assign_values (3 coordinate values)

Assign three values to a geometry (usually a 3D point)

Synopsis

```
template<typename Geometry, typename Type>
void assign_values(Geometry & geometry, Type const & c1, Type const & c2, Type const & c3)
```

Parameters

Type	Concept	Name	Description
Geometry &	Any type fulfilling a Geometry Concept	geometry	A model of the specified concept
Type const &	numerical type (int, double, ttmath, ...) to specify the co-ordinates	c1	First coordinate (usually x-coordinate)
Type const &	numerical type (int, double, ttmath, ...) to specify the co-ordinates	c2	Second coordinate (usually y-coordinate)
Type const &	numerical type (int, double, ttmath, ...) to specify the co-ordinates	c3	Third coordinate (usually z-coordinate)

Header

```
#include <boost/geometry/algorithms/detail/assign_values.hpp>
```

Example

Use assign to set three coordinates of a 3D point

```
#include <iostream>
#include <iomanip>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/point.hpp>

int main()
{
    boost::geometry::model::point<double, 3, boost::geometry::cs::cartesian> p;
    boost::geometry::assign_values(p, 1.2345, 2.3456, 3.4567);

    std::cout << boost::geometry::dsv(p) << std::endl;

    return 0;
}
```

Output:

```
(1.2345, 2.3456, 3.4567)
```

See also

- [make](#)

assign_zero

assign zero values to a box, point

Description

The assign_zero function initializes a 2D or 3D point or box with coordinates of zero

Synopsis

```
template<typename Geometry>
void assign_zero(Geometry & geometry)
```

Parameters

Type	Concept	Name	Description
Geometry &	Any type fulfilling a Geometry Concept	geometry	A model of the specified concept

Header

Either

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

Or

```
#include <boost/geometry/algorithms/assign.hpp>
```

append

Appends one or more points to a linestring, ring, polygon, multi-geometry.

Synopsis

```
template<typename Geometry, typename RangeOrPoint>
void append(Geometry & geometry, RangeOrPoint const & range_or_point, int ring_index = -1, int multi_index = 0)
```

Parameters

Type	Concept	Name	Description
Geometry &	Any type fulfilling a Geometry Concept	geometry	A model of the specified concept
RangeOrPoint const &	Either a range or a point, full-filling Boost.Range concept or Boost.Geometry Point Concept	range_or_point	The point or range to add
int		ring_index	The index of the ring in case of a polygon: exterior ring (-1, the default) or interior ring index
int		multi_index	Reserved for multi polygons or multi linestrings

Header

Either

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

Or

```
#include <boost/geometry/algorithms/append.hpp>
```

Conformance

The function append is not defined by OGC.

Supported geometries

	Point	Range
Point	✓	✓
Segment	✓	✓
Box	✓	✓
Linestring	✓	✓
Ring	✓	✓
Polygon	✓	✓
MultiPoint	✓	✓
MultiLinestring	✗	✗
MultiPolygon	✗	✗

Behavior

Case	Behavior
Point, Rectangle, Segment	Compiles, but no action
Linestring	Appends point or range to the end of the linestring
Ring	Appends point or range to the end of the ring (without explicitly closing it)
Polygon	Appends point or range to the end of the polygon (without explicitly closing it), either the exterior ring (the default) or specify a zero-based index for one of the interior rings. In the last case, the interior rings are not resized automatically, so ensure that the zero-based index is smaller than the number of interior rings

Complexity

Linear

Example

Shows usage of Boost.Geometry's append to append a point or a range to a polygon

```

#include <iostream>

#include <boost/assign.hpp>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/polygon.hpp>
#include <boost/geometry/geometries/adapted/boost_tuple.hpp>

BOOST_GEOMETRY_REGISTER_BOOST_TUPLE_CS(cs::cartesian)

int main()
{
    using boost::assign::tuple_list_of;
    using boost::make_tuple;
    using boost::geometry::append;

    typedef boost::geometry::model::polygon<boost::tuple<int, int> > polygon;

    polygon poly;

    // Append a range
    append(poly, tuple_list_of(0, 0)(0, 10)(11, 11)(10, 0)); ❶
    // Append a point (in this case the closing point)
    append(poly, make_tuple(0, 0));

    // Create an interior ring (append does not do this automatically)
    boost::geometry::interior_rings(poly).resize(1);

    // Append a range to the interior ring
    append(poly, tuple_list_of(2, 2)(2, 5)(6, 6)(5, 2), 0); ❷
    // Append a point to the first interior ring
    append(poly, make_tuple(2, 2), 0);

    std::cout << boost::geometry::dsv(poly) << std::endl;

    return 0;
}

```

- ❶ `tuple_list_of` delivers a range and can therefore be used in `boost::geometry::append`
- ❷ The last parameter `ring_index 0` denotes the first interior ring

Output:

```
((0, 0), (0, 10), (11, 11), (10, 0), (0, 0)), ((2, 2), (2, 5), (6, 6), (5, 2), (2, 2)))
```

See also

- [assign](#)

buffer

buffer

Calculates the buffer of a geometry.

Description

The free function `buffer` calculates the buffer (a polygon being the spatial point set collection within a specified maximum distance from a geometry) of a geometry.

Synopsis

```
template<typename Input, typename Output, typename Distance>
void buffer(Input const & geometry_in, Output & geometry_out, Distance const & distance, Distance const & chord_length = -1)
```

Parameters

Type	Concept	Name	Description
Input const &	Any type fulfilling a Geometry Concept	geometry_in	A model of the specified concept
Output &	Any type fulfilling a Geometry Concept	geometry_out	A model of the specified concept
Distance const &	numerical type (int, double, ttmath, ...)	distance	The distance to be used for the buffer
Distance const &	numerical type (int, double, ttmath, ...)	chord_length	(optional) The length of the chord's in the generated arcs around points or bends

Header

Either

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

Or

```
#include <boost/geometry/algorithms/buffer.hpp>
```

Conformance

The function `buffer` implements function `Buffer` from the [OGC Simple Feature Specification](#).



Note

The current implementation only enlarges a box, which is not defined by OGC. A next version of the library will contain a more complete implementation

Behavior

Case	Behavior
Rectangle/Rectangle	Returns a new rectangular box, enlarged with the specified distance. It is allowed that "geometry_out" the same object as "geometry_in"

return_buffer

Calculates the buffer of a geometry.

Description

The free function `return_buffer` calculates the buffer (a polygon being the spatial point set collection within a specified maximum distance from a geometry) of a geometry. This version with the `return_` prefix returns the buffer, and a template parameter must therefore be specified in the call..

Synopsis

```
template<typename Output, typename Input, typename T, >
Output return_buffer(Input const & geometry, T const & distance, T const & chord_length = -1)
```

Parameters

Type	Concept	Name	Description
Output	Any type fulfilling a Geometry Concept	-	Must be specified
Distance	numerical type (int, double, ttmath, ...)	-	Must be specified
Input const &	Any type fulfilling a Geometry Concept	geometry	A model of the specified concept
T const &		distance	The distance to be used for the buffer
T const &		chord_length	(optional) The length of the chord's in the generated arcs around points or bends

Returns

The calculated buffer

Header

Either

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

Or

```
#include <boost/geometry/algorithms/buffer.hpp>
```

centroid

centroid (with strategy)

Calculates the centroid of a geometry using the specified strategy.

Description

The free function `centroid` calculates the geometric center (or: center of mass) of a geometry. Reasons to specify a strategy include: use another coordinate system for calculations; construct the strategy beforehand (e.g. with the radius of the Earth); select a strategy when there are more than one available for a calculation.

Synopsis

```
template<typename Geometry, typename Point, typename Strategy>
void centroid(Geometry const & geometry, Point & c, Strategy const & strategy)
```

Parameters

Type	Concept	Name	Description
Geometry const &	Any type fulfilling a Geometry Concept	geometry	A model of the specified concept
Point &	Any type fulfilling a Point Concept	c	A model of the specified Point Concept which is set to the centroid
Strategy const &	Any type fulfilling a Centroid Strategy Concept	strategy	The strategy which will be used for centroid calculations

Header

Either

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

Or

```
#include <boost/geometry/algorithms/centroid.hpp>
```

Conformance

The function centroid implements function Centroid from the [OGC Simple Feature Specification](#).

Behavior

Case	Behavior
Point	Returns the point itself as the centroid
Multi Point	Calculates centroid (based on average)
linear (e.g. linestring)	Calculates centroid (based on weighted length)
areal (e.g. polygon)	Calculates centroid
Empty (e.g. polygon without points)	Throws a centroid_exception
Cartesian	Implemented
Spherical	Calculates the centroid as if based on Cartesian coordinates

Supported geometries

	2D	3D
Point	✔	✔
Segment	✔	✔
Box	✔	✔
Linestring	✔	✘
Ring	✔	✘
Polygon	✔	✘
MultiPoint	✔	✔
MultiLinestring	✔	✘
MultiPolygon	✔	✘

Complexity

Linear

Available Strategies

- Bashein Detmer (cartesian)

centroid

Calculates the centroid of a geometry.

Description

The free function centroid calculates the geometric center (or: center of mass) of a geometry. It uses the default strategy, based on the coordinate system of the geometry.

Synopsis

```
template<typename Geometry, typename Point>
void centroid(Geometry const & geometry, Point & c)
```

Parameters

Type	Concept	Name	Description
Geometry const &	Any type fulfilling a Geometry Concept	geometry	A model of the specified concept
Point &	Any type fulfilling a Point Concept	c	The calculated centroid will be assigned to this point reference

Header

Either

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

Or

```
#include <boost/geometry/algorithms/centroid.hpp>
```

Conformance

The function `centroid` implements function `Centroid` from the [OGC Simple Feature Specification](#).

Behavior

Case	Behavior
Point	Returns the point itself as the centroid
Multi Point	Calculates centroid (based on average)
linear (e.g. linestring)	Calculates centroid (based on weighted length)
areal (e.g. polygon)	Calculates centroid
Empty (e.g. polygon without points)	Throws a centroid_exception
Cartesian	Implemented
Spherical	Calculates the centroid as if based on Cartesian coordinates

Supported geometries

	2D	3D
Point	✓	✓
Segment	✓	✓
Box	✓	✓
Linestring	✓	✗
Ring	✓	✗
Polygon	✓	✗
MultiPoint	✓	✓
MultiLinestring	✓	✗
MultiPolygon	✓	✗

Complexity

Linear

Example

Shows calculation of a centroid of a polygon

```
#include <iostream>
#include <list>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/point_xy.hpp>
#include <boost/geometry/geometries/polygon.hpp>

#include <boost/geometry/io/wkt/wkt.hpp>

int main()
{
    typedef boost::geometry::model::d2::point_xy<double> point_type;
    typedef boost::geometry::model::polygon<point_type> polygon_type;

    polygon_type poly;
    boost::geometry::read_wkt(
        "POLYGON((2 1.3,2.4 1.7,2.8 1.8,3.4 1.2,3.7 1.6,3.4 2,4.1 3,5.3 2.6,5.4 1.2,4.9 0.8,2.9 0.7,2 1.3)"
        "(4.0 2.0, 4.2 1.4, 4.8 1.9, 4.4 2.2, 4.0 2.0))", poly);

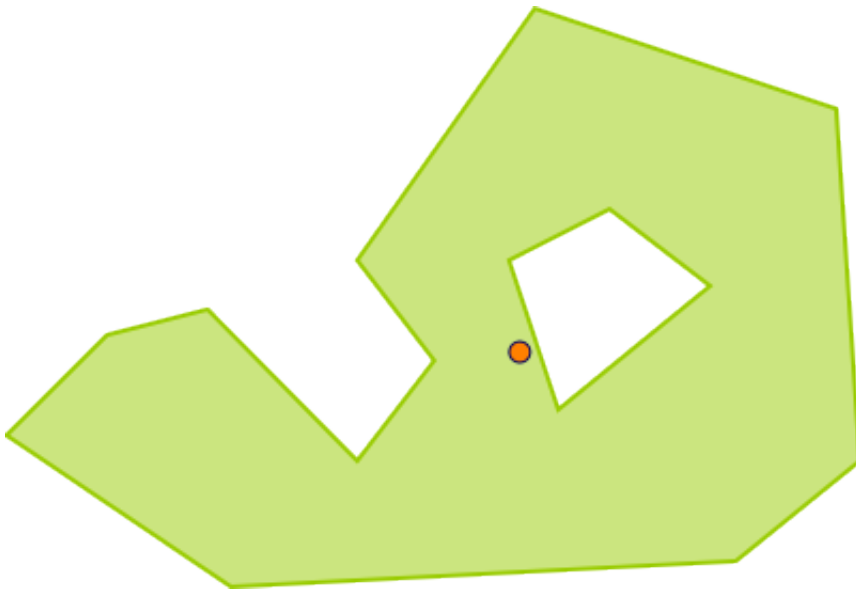
    point_type p;
    boost::geometry::centroid(poly, p);

    std::cout << "centroid: " << boost::geometry::dsv(p) << std::endl;

    return 0;
}
```

Output:

centroid: (4.04663, 1.6349)



Note that the centroid might be located in a hole or outside a polygon, easily.

return_centroid

Calculates the centroid of a geometry.

Description

The free function centroid calculates the geometric center (or: center of mass) of a geometry. This version with the return_ prefix returns the centroid, and a template parameter must therefore be specified in the call..

Synopsis

```
template<typename Point, typename Geometry>
Point return_centroid(Geometry const & geometry)
```

Parameters

Type	Concept	Name	Description
Point	Any type fulfilling a Point Concept	-	Must be specified
Geometry const &	Any type fulfilling a Geometry Concept	geometry	A model of the specified concept

Returns

The calculated centroid

Header

Either

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

Or

```
#include <boost/geometry/algorithms/centroid.hpp>
```

Conformance

The function centroid implements function Centroid from the [OGC Simple Feature Specification](#).

Behavior

Case	Behavior
Point	Returns the point itself as the centroid
Multi Point	Calculates centroid (based on average)
linear (e.g. linestring)	Calculates centroid (based on weighted length)
areal (e.g. polygon)	Calculates centroid
Empty (e.g. polygon without points)	Throws a centroid_exception
Cartesian	Implemented
Spherical	Calculates the centroid as if based on Cartesian coordinates

Supported geometries

	2D	3D
Point		
Segment		
Box		
Linestring		
Ring		
Polygon		
MultiPoint		
MultiLinestring		
MultiPolygon		

Complexity

Linear

return_centroid (with strategy)

Calculates the centroid of a geometry using the specified strategy.

Description

The free function centroid calculates the geometric center (or: center of mass) of a geometry. This version with the return_ prefix returns the centroid, and a template parameter must therefore be specified in the call.. Reasons to specify a strategy include: use another coordinate system for calculations; construct the strategy beforehand (e.g. with the radius of the Earth); select a strategy when there are more than one available for a calculation.

Synopsis

```
template<typename Point, typename Geometry, typename Strategy>
Point return_centroid(Geometry const & geometry, Strategy const & strategy)
```

Parameters

Type	Concept	Name	Description
Point	Any type fulfilling a Point Concept	-	Must be specified
Geometry const &	Any type fulfilling a Geometry Concept	geometry	A model of the specified concept
Strategy const &	Any type fulfilling a centroid Strategy Concept	strategy	The strategy which will be used for centroid calculations

Returns

The calculated centroid

Header

Either

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

Or

```
#include <boost/geometry/algorithms/centroid.hpp>
```

Conformance

The function centroid implements function Centroid from the [OGC Simple Feature Specification](#).

Behavior

Case	Behavior
Point	Returns the point itself as the centroid
Multi Point	Calculates centroid (based on average)
linear (e.g. linestring)	Calculates centroid (based on weighted length)
areal (e.g. polygon)	Calculates centroid
Empty (e.g. polygon without points)	Throws a centroid_exception
Cartesian	Implemented
Spherical	Calculates the centroid as if based on Cartesian coordinates

Supported geometries

	2D	3D
Point	✓	✓
Segment	✓	✓
Box	✓	✓
Linestring	✓	✗
Ring	✓	✗
Polygon	✓	✗
MultiPoint	✓	✓
MultiLinestring	✓	✗
MultiPolygon	✓	✗

Complexity

Linear

Available Strategies

- Bashein Detmer (cartesian)

clear

Clears a linestring, ring or polygon (exterior+interiors) or multi*.

Description

Generic function to clear a geometry. All points will be removed from the collection or collections making up the geometry. In most cases this is equivalent to the `.clear()` method of a `std::vector<...>`. In the case of a polygon, this clear functionality is automatically called for the exterior ring, and for the interior ring collection. In the case of a point, boxes and segments, nothing will happen.

Synopsis

```
template<typename Geometry>
void clear(Geometry & geometry)
```

Parameters

Type	Concept	Name	Description
Geometry &	Any type fulfilling a Geometry Concept	geometry	A model of the specified concept which will be cleared

Header

Either

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

Or










```
#include <boost/geometry/algorithms/clear.hpp>
```

Conformance

The function `clear` is not defined by OGC.

The function `clear` conforms to the `clear()` method of the C++ std-library.

Supported geometries

Geometry	Status
Point	
Segment	
Box	
Linestring	
Ring	
Polygon	
MultiPoint	
MultiLinestring	
MultiPolygon	

Behavior

Case	Behavior
Point	Nothing happens, geometry is unchanged
Segment	Nothing happens, geometry is unchanged
Rectangle	Nothing happens, geometry is unchanged
Linestring	Linestring is cleared
Ring	Ring is cleared
Polygon	The exterior ring is cleared and all interior rings are removed
Multi Point	Multi Point is cleared
Multi Linestring	Multi Linestring is cleared
Multi Polygon	Multi Polygon is cleared

Complexity

Constant

Example

Shows how to clear a ring or polygon

```

#include <iostream>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/polygon.hpp>
#include <boost/geometry/geometries/ring.hpp>
#include <boost/geometry/geometries/adapted/boost_tuple.hpp>

BOOST_GEOMETRY_REGISTER_BOOST_TUPLE_CS(cs::cartesian)

#include <boost/assign.hpp>

int main()
{
    using boost::assign::tuple_list_of;

    typedef boost::tuple<float, float> point;
    typedef boost::geometry::model::polygon<point> polygon;
    typedef boost::geometry::model::ring<point> ring;

    polygon poly;

    // Fill the polygon (using its own methods + Boost.Assign)
    poly.outer() = tuple_list_of(0, 0)(0, 9)(10, 10)(0, 0);
    poly.inners().push_back(tuple_list_of(1, 2)(4, 6)(2, 8)(1, 2));

    std::cout << boost::geometry::dsv(poly) << std::endl;
    boost::geometry::clear(poly);
    std::cout << boost::geometry::dsv(poly) << std::endl;

    // Create a ring using Boost.Assign
    ring r = tuple_list_of(0, 0)(0, 9)(8, 8)(0, 0);

    std::cout << boost::geometry::dsv(r) << std::endl;
    boost::geometry::clear(r);
    std::cout << boost::geometry::dsv(r) << std::endl;

    return 0;
}

```

Output:

```

((0, 0), (0, 10), (11, 11), (0, 0)), ((0, 0), (0, 10), (11, 11), (0, 0)))
(())
((0, 0), (0, 9), (8, 8), (0, 0))
( )

```

convert

Converts one geometry to another geometry.

Description

The convert algorithm converts one geometry, e.g. a BOX, to another geometry, e.g. a RING. This only if it is possible and applicable. If the point-order is different, or the closure is different between two geometry types, it will be converted correctly by explicitly reversing the points or closing or opening the polygon rings.

Synopsis

```

template<typename Geometry1, typename Geometry2>
void convert(Geometry1 const & geometry1, Geometry2 & geometry2)

```

Parameters

Type	Concept	Name	Description
Geometry1 const &	Any type fulfilling a Geometry Concept	geometry1	A model of the specified concept (source)
Geometry2 &	Any type fulfilling a Geometry Concept	geometry2	A model of the specified concept (target)

Header

Either

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

Or

```
#include <boost/geometry/algorithms/convert.hpp>
```

Conformance

The function `convert` is not defined by OGC.

Supported geometries

	Point	Segment	Box	Line-string	Ring	Polygon	Multi-Point	MultiLine-string	MultiPolygon
Point	✓	✗	✗	✗	✗	✗	✗	✗	✗
Segment	✗	✓	✗	✗	✗	✗	✗	✗	✗
Box	✓	✗	✓	✗	✗	✗	✗	✗	✗
Linestring	✗	✓	✗	✓	✗	✗	✗	✗	✗
Ring	✗	✗	✓	✗	✓	✓	✗	✗	✗
Polygon	✗	✗	✓	✗	✓	✓	✗	✗	✗
Multi-Point	✓	✗	✗	✗	✗	✗	✓	✗	✗
MultiLine-string	✗	✓	✗	✓	✗	✗	✗	✓	✗
MultiPolygon	✗	✗	✓	✗	✓	✓	✗	✗	✓



Note

In this status matrix above: columns are source types and rows are target types. So a box can be converted to a ring, polygon or multi-polygon, but not vice versa.

Complexity

Linear

Example

Shows how to convert a geometry into another geometry

```
#include <iostream>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/box.hpp>
#include <boost/geometry/geometries/point_xy.hpp>
#include <boost/geometry/geometries/polygon.hpp>
#include <boost/geometry/geometries/adapted/boost_tuple.hpp>

BOOST_GEOMETRY_REGISTER_BOOST_TUPLE_CS(cs::cartesian)

int main()
{
    typedef boost::geometry::model::d2::point_xy<double> point;
    typedef boost::geometry::model::box<point> box;
    typedef boost::geometry::model::polygon<point> polygon;

    point p1(1, 1);
    box bx = boost::geometry::make<box>(1, 1, 2, 2);

    // Assign a box to a polygon (conversion box->poly)
    polygon poly;
    boost::geometry::convert(bx, poly);

    // Convert a point to another point type (conversion of point-type)
    boost::tuple<double, double> p2;
    boost::geometry::convert(p1, p2); // source -> target

    using boost::geometry::dsv;
    std::cout
        << "box: " << dsv(bx) << std::endl
        << "polygon: " << dsv(poly) << std::endl
        << "point: " << dsv(p1) << std::endl
        << "point tuples: " << dsv(p2) << std::endl
        ;

    return 0;
}
```

Output:

```
box: ((1, 1), (2, 2))
polygon: (((1, 1), (1, 2), (2, 2), (2, 1), (1, 1)))
point: (1, 1)
point tuples: (1, 1)
```

See also

- [assign](#)



Note

convert is modelled as source -> target (where assign is modelled as target := source)

convex_hull

Calculates the convex hull of a geometry.

Description

The free function `convex_hull` calculates the convex hull of a geometry.

Synopsis

```
template<typename Geometry, typename OutputGeometry, , >
void convex_hull(Geometry const & geometry, OutputGeometry & hull)
```

Parameters

Type	Concept	Name	Description
Geometry1	Any type fulfilling a Geometry Concept	-	Must be specified
Geometry2	Any type fulfilling a Geometry Concept	-	Must be specified
Geometry const &		geometry	A model of the specified concept, input geometry
OutputGeometry &		hull	A model of the specified concept which is set to the convex hull

Header

Either

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










Or

```
#include <boost/geometry/algorithms/convex_hull.hpp>
```

Conformance

The function `convex_hull` implements function `ConvexHull()` from the [OGC Simple Feature Specification](#).

Supported geometries

Geometry	Status
Point	
Segment	
Box	
Linestring	
Ring	
Polygon	
MultiPoint	
MultiLinestring	
MultiPolygon	

Complexity

Logarithmic

Example

Shows how to generate the `convex_hull` of a geometry

```

#include <iostream>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/polygon.hpp>
#include <boost/geometry/geometries/adapted/boost_tuple.hpp>

BOOST_GEOMETRY_REGISTER_BOOST_TUPLE_CS(cs::cartesian)

int main()
{
    typedef boost::tuple<double, double> point;
    typedef boost::geometry::model::polygon<point> polygon;

    polygon poly;
    boost::geometry::read_wkt("polygon((2.0 1.3, 2.4 1.7, 2.8 1.8, 3.4 1.2, 3.7 1.6,3.4 2.0, ↵
4.1 3.0"
        ", 5.3 2.6, 5.4 1.2, 4.9 0.8, 2.9 0.7,2.0 1.3))", poly);

    polygon hull;
    boost::geometry::convex_hull(poly, hull);

    using boost::geometry::dsv;
    std::cout
        << "polygon: " << dsv(poly) << std::endl
        << "hull: " << dsv(hull) << std::endl
        ;

    return 0;
}

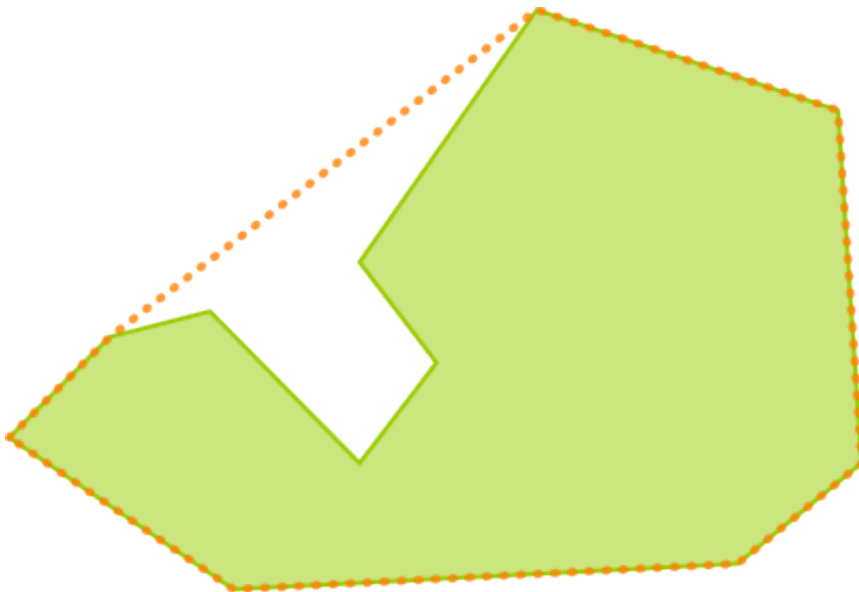
```

Output:

```

polygon: (((2, 1.3), (2.4, 1.7), (2.8, 1.8), (3.4, 1.2), (3.7, 1.6), (3.4, 2), (4.1, 3), (5.3, ↵
2.6), (5.4, 1.2), (4.9, 0.8), (2.9, 0.7), (2, 1.3)))
hull: (((2, 1.3), (2.4, 1.7), (4.1, 3), (5.3, 2.6), (5.4, 1.2), (4.9, 0.8), (2.9, 0.7), (2, 1.3)))

```



correct

Corrects a geometry.

Description

Corrects a geometry: all rings which are wrongly oriented with respect to their expected orientation are reversed. To all rings which do not have a closing point and are typed as they should have one, the first point is appended. Also boxes can be corrected.

Synopsis

```
template<typename Geometry>
void correct(Geometry & geometry)
```

Parameters

Type	Concept	Name	Description
Geometry &	Any type fulfilling a Geometry Concept	geometry	A model of the specified concept which will be corrected if necessary

Header

Either

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

Or

```
#include <boost/geometry/algorithms/correct.hpp>
```

Conformance

The function correct is not defined by OGC.

Supported geometries

Geometry	Status
Point	✓
Segment	✓
Box	✓
Linestring	✓
Ring	✓
Polygon	✓
MultiPoint	✓
MultiLinestring	✓
MultiPolygon	✓

Behavior

Case	Behavior
Ring	Ring is corrected
Polygon	Polygon is corrected
Multi Polygon	Multi Polygon is corrected
Rectangle	Rectangle is corrected with respect to minimal and maximal corners
Other geometries	Nothing happens, geometry is unchanged

Complexity

Linear

Example

Shows how to correct a polygon with respect to its orientation and closure

```

#include <iostream>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/polygon.hpp>
#include <boost/geometry/geometries/adapted/boost_tuple.hpp>

BOOST_GEOMETRY_REGISTER_BOOST_TUPLE_CS(cs::cartesian)

#include <boost/assign.hpp>

int main()
{
    using boost::assign::tuple_list_of;

    typedef boost::geometry::model::polygon
    <
        boost::tuple<int, int>
    > clockwise_closed_polygon;

    clockwise_closed_polygon cwcp;

    // Fill it counterclockwise (so wrongly), forgetting the closing point
    boost::geometry::exterior_ring(cwcp) = tuple_list_of(0, 0)(10, 10)(0, 9);

    // Add a counterclockwise closed inner ring (this is correct)
    boost::geometry::interior_rings(cwcp).push_back(tuple_list_of(1, 2)(4, 6)(2, 8)(1, 2));

    // Its area should be negative (because of wrong orientation)
    // and wrong (because of omitted closing point)
    double area_before = boost::geometry::area(cwcp);

    // Correct it!
    boost::geometry::correct(cwcp);

    // Check its new area
    double area_after = boost::geometry::area(cwcp);

    // And output it
    std::cout << boost::geometry::dsv(cwcp) << std::endl;
    std::cout << area_before << " -> " << area_after << std::endl;

    return 0;
}

```

Output:

```

(((0, 0), (0, 9), (10, 10), (0, 0)), ((1, 2), (4, 6), (2, 8), (1, 2)))
-7 -> 38

```

covered_by

covered_by

Checks if the first geometry is inside or on border the second geometry.

Description

The free function `covered_by` checks if the first geometry is inside or on border the second geometry.

Synopsis

```
template<typename Geometry1, typename Geometry2>
bool covered_by(Geometry1 const & geometry1, Geometry2 const & geometry2)
```

Parameters

Type	Concept	Name	Description
Geometry1 const &	Any type fulfilling a Geometry Concept	geometry1	A model of the specified concept which might be inside or on the border of the second geometry
Geometry2 const &	Any type fulfilling a Geometry Concept	geometry2	A model of the specified concept which might cover the first geometry

Returns

true if geometry1 is inside of or on the border of geometry2, else false

Header

Either

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

Or

```
#include <boost/geometry/algorithms/covered_by.hpp>
```

Conformance

The function covered_by is not defined by OGC.



Note

Both PostGIS and Oracle contain an algorithm with the same name and the same functionality. See the [PostGIS documentation](#).

Supported geometries

	Point	Segment	Box	Lines - tring	Ring	Polygon	M u l t i - Point	MultiLin- estring	MultiPoly- gon
Point	✗	✗	✗	✗	✗	✗	✗	✗	✗
Segment	✗	✗	✗	✗	✗	✗	✗	✗	✗
Box	✓	✗	✓	✗	✗	✗	✗	✗	✗
Linestring	✗	✗	✗	✗	✗	✗	✗	✗	✗
Ring	✓	✗	✗	✗	✗	✗	✗	✗	✗
Polygon	✓	✗	✗	✗	✗	✗	✗	✗	✗
M u l t i - Point	✗	✗	✗	✗	✗	✗	✗	✗	✗
MultiLin- estring	✗	✗	✗	✗	✗	✗	✗	✗	✗
MultiPoly- gon	✓	✗	✗	✗	✗	✗	✗	✗	✗

**Note**

In this status matrix above: columns are types of first parameter and rows are types of second parameter. So a point can be checked to be covered by a polygon, but not vice versa.

Complexity

Linear

See also

- [within](#)

**Note**

The difference with the `within` algorithm is that this algorithm checks the border by default

covered_by (with strategy)

Checks if the first geometry is inside or on border the second geometry using the specified strategy.

Description

The free function `covered_by` checks if the first geometry is inside or on border the second geometry, using the specified strategy. Reasons to specify a strategy include: use another coordinate system for calculations; construct the strategy beforehand (e.g. with the radius of the Earth); select a strategy when there are more than one available for a calculation.

Synopsis

```
template<typename Geometry1, typename Geometry2, typename Strategy>
bool covered_by(Geometry1 const & geometry1, Geometry2 const & geometry2, Strategy const & strategy)
```

Parameters

Type	Concept	Name	Description
Geometry1 const &	Any type fulfilling a Geometry Concept	geometry1	A model of the specified concept which might be inside or on the border of the second geometry
Geometry2 const &	Any type fulfilling a Geometry Concept	geometry2	A model of the specified concept which might cover the first geometry
Strategy const &		strategy	strategy to be used

Returns

true if geometry1 is inside of or on the border of geometry2, else false

Header

Either

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

Or

```
#include <boost/geometry/algorithms/covered_by.hpp>
```

Conformance

The function covered_by is not defined by OGC.



Note

Both PostGIS and Oracle contain an algorithm with the same name and the same functionality. See the [PostGIS documentation](#).

Supported geometries

	Point	Segment	Box	Line-string	Ring	Polygon	Multi-Point	MultiLine-string	MultiPolygon
Point	✗	✗	✗	✗	✗	✗	✗	✗	✗
Segment	✗	✗	✗	✗	✗	✗	✗	✗	✗
Box	✓	✗	✓	✗	✗	✗	✗	✗	✗
Linestring	✗	✗	✗	✗	✗	✗	✗	✗	✗
Ring	✓	✗	✗	✗	✗	✗	✗	✗	✗
Polygon	✓	✗	✗	✗	✗	✗	✗	✗	✗
Multi-Point	✗	✗	✗	✗	✗	✗	✗	✗	✗
MultiLine-string	✗	✗	✗	✗	✗	✗	✗	✗	✗
MultiPolygon	✓	✗	✗	✗	✗	✗	✗	✗	✗

**Note**

In this status matrix above: columns are types of first parameter and rows are types of second parameter. So a point can be checked to be covered by a polygon, but not vice versa.

Complexity

Linear

See also

- [within](#)

**Note**

The difference with the `within` algorithm is that this algorithm checks the border by default

difference

Description

Calculate the difference of two geometries

The free function `difference` calculates the spatial set theoretic difference of two geometries.

Synopsis

```
template<typename Geometry1, typename Geometry2, typename Collection>
void difference(Geometry1 const & geometry1, Geometry2 const & geometry2, Collection & output_collection)
```

Parameters

Type	Concept	Name	Description
Geometry1 const &	Any type fulfilling a Geometry Concept	geometry1	A model of the specified concept
Geometry2 const &	Any type fulfilling a Geometry Concept	geometry2	A model of the specified concept
Collection &	output collection, either a multi-geometry, or a <code>std::vector<Geometry></code> / <code>std::deque<Geometry></code> etc	output_collection	the output collection

Header

Either

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

Or

```
#include <boost/geometry/algorithms/difference.hpp>
```

Conformance

The function difference implements function Difference from the [OGC Simple Feature Specification](#).

Behavior

Case	Behavior
areal (e.g. polygon)	All combinations of: box, ring, polygon, multi_polygon
linear (e.g. linestring) / areal (e.g. polygon)	A combinations of a (multi) linestring with a (multi) polygon results in a collection of linestrings
Other geometries	Not yet supported in this version
Spherical	Not yet supported in this version
Three dimensional	Not yet supported in this version



Note

Check the [Polygon Concept](#) for the rules that polygon input for this algorithm should fulfill

Example

Shows how to subtract one polygon from another polygon

```
#include <iostream>
#include <list>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/point_xy.hpp>
#include <boost/geometry/geometries/polygon.hpp>
#include <boost/geometry/io/wkt/wkt.hpp>

#include <boost/foreach.hpp>

int main()
{
    typedef boost::geometry::model::polygon<boost::geometry::model::d2::point_xy<double> > polygon;

    polygon green, blue;

    boost::geometry::read_wkt(
        "POLYGON((2 1.3,2.4 1.7,2.8 1.8,3.4 1.2,3.7 1.6,3.4 2,4.1 3,5.3 2.6,5.4 1.2,4.9 0.8,2.9 ↵
0.7,2 1.3))"
        "(4.0 2.0, 4.2 1.4, 4.8 1.9, 4.4 2.2, 4.0 2.0))", green);

    boost::geometry::read_wkt(
        "POLYGON((4.0 -0.5 , 3.5 1.0 , 2.0 1.5 , 3.5 2.0 , 4.0 3.5 , 4.5 2.0 , 6.0 1.5 , 4.5 ↵
1.0 , 4.0 -0.5))", blue);

    std::list<polygon> output;
    boost::geometry::difference(green, blue, output);

    int i = 0;
    std::cout << "green - blue:" << std::endl;
    BOOST_FOREACH(polygon const& p, output)
    {
        std::cout << i++ << ": " << boost::geometry::area(p) << std::endl;
    }

    output.clear();
    boost::geometry::difference(blue, green, output);

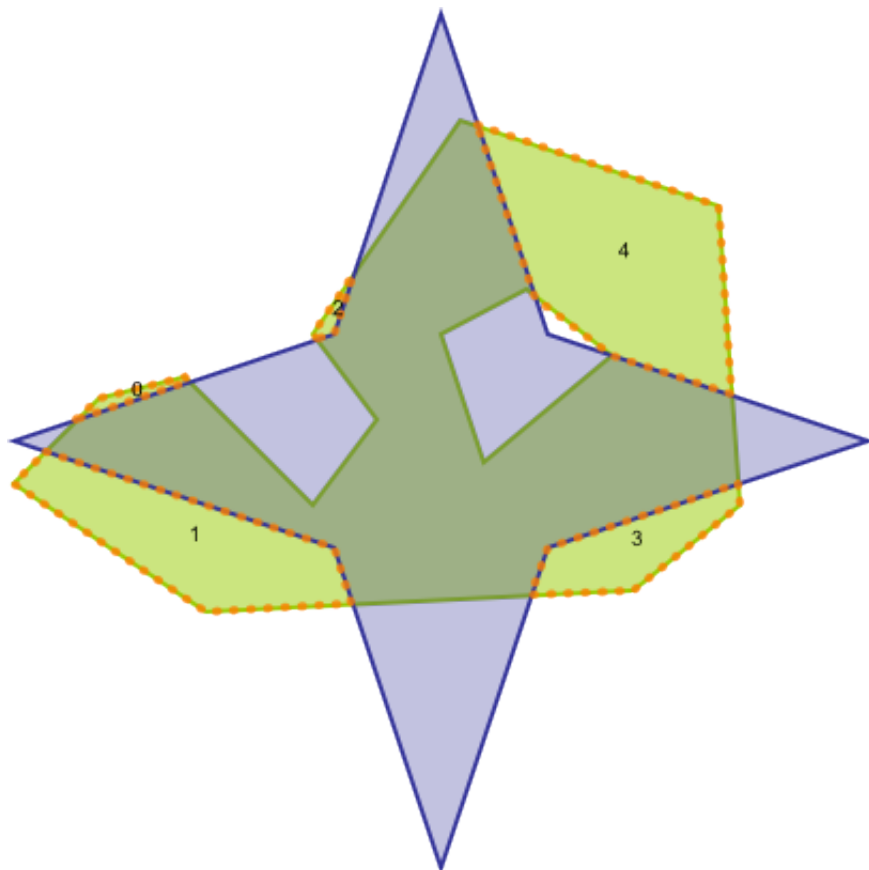
    i = 0;
    std::cout << "blue - green:" << std::endl;
    BOOST_FOREACH(polygon const& p, output)
    {
        std::cout << i++ << ": " << boost::geometry::area(p) << std::endl;
    }

    return 0;
}
```

Output:

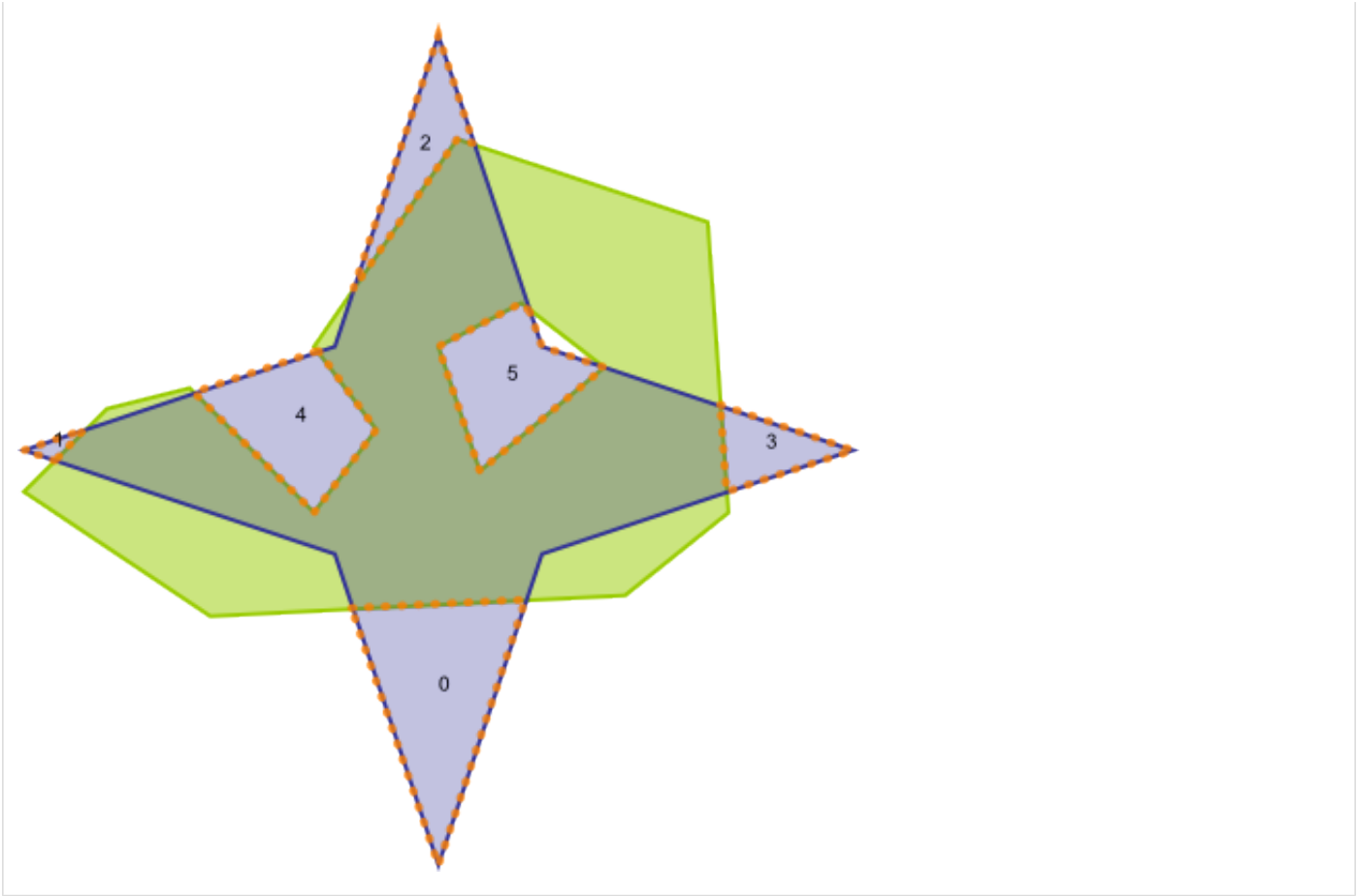
green - blue:

0: 0.02375
1: 0.542951
2: 0.0149697
3: 0.226855
4: 0.839424



blue - green:

0: 0.525154
1: 0.015
2: 0.181136
3: 0.128798
4: 0.340083
5: 0.307778



See also

- [sym_difference](#) (symmetric difference)
- [intersection](#)
- [union](#)

disjoint

Checks if two geometries are disjoint.

Synopsis

```
template<typename Geometry1, typename Geometry2>
bool disjoint(Geometry1 const & geometry1, Geometry2 const & geometry2)
```

Parameters

Type	Concept	Name	Description
Geometry1 const &	Any type fulfilling a Geometry Concept	geometry1	A model of the specified concept
Geometry2 const &	Any type fulfilling a Geometry Concept	geometry2	A model of the specified concept

Returns

Returns true if two geometries are disjoint

Header

Either

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

Or

```
#include <boost/geometry/algorithms/disjoint.hpp>
```

Conformance

The function disjoint implements function Disjoint from the [OGC Simple Feature Specification](#).

distance

comparable_distance

Calculate the comparable distance measurement of two geometries.

Description

The free function comparable_distance does not necessarily calculate the distance, but it calculates a distance measure such that two distances are comparable to each other. For example: for the Cartesian coordinate system, Pythagoras is used but the square root is not taken, which makes it faster and the results of two point pairs can still be compared to each other.

Synopsis

```
template<typename Geometry1, typename Geometry2>
default_distance_result<Geometry1, Geometry2>::type comparable_distance(Geometry1 const & geometry1,
Geometry2 const & geometry2)
```

Parameters

Type	Concept	Name	Description
Geometry1 const &	first geometry type	geometry1	A model of the specified concept
Geometry2 const &	second geometry type	geometry2	A model of the specified concept

Returns

The calculated comparable distance

Header

Either

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

Or

```
#include <boost/geometry/algorithms/comparable_distance.hpp>
```

Conformance

The function `comparable_distance` is not defined by OGC.

Behaviour

There is no (not yet) version with a strategy.

It depends on the coordinate system of the geometry's point type if there is a strategy available which can determine (more efficient than the standard strategy) a measure of comparable distance.

Complexity

Linear

Example

Shows how to efficiently get the closest point

```
#include <iostream>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/point_xy.hpp>

#include <boost/numeric/conversion/bounds.hpp>
#include <boost/foreach.hpp>

int main()
{
    typedef boost::geometry::model::d2::point_xy<double> point_type;

    point_type p(1.4, 2.6);

    std::vector<point_type> v;
    for (double x = 0.0; x <= 4.0; x++)
    {
        for (double y = 0.0; y <= 4.0; y++)
        {
            v.push_back(point_type(x, y));
        }
    }

    point_type min_p;
    double min_d = boost::numeric::bounds<double>::highest();
    BOOST_FOREACH(point_type const& pv, v)
    {
        double d = boost::geometry::comparable_distance(p, pv);
        if (d < min_d)
        {
            min_d = d;
            min_p = pv;
        }
    }

    std::cout
        << "Closest: " << boost::geometry::dsv(min_p) << std::endl
        << "At: " << boost::geometry::distance(p, min_p) << std::endl;

    return 0;
}
```

Output:

```
Closest: (1, 3)
At: 0.565685
```

distance

Calculate the distance of two geometries.

Description

The default strategy is used, corresponding to the coordinate system of the geometries

Synopsis

```
template<typename Geometry1, typename Geometry2>
default_distance_result<Geometry1, Geometry2>::type distance(Geometry1 const & geometry1, Geo-
metry2 const & geometry2)
```

Parameters

Type	Concept	Name	Description
Geometry1 const &	Any type fulfilling a Geometry Concept	geometry1	A model of the specified concept
Geometry2 const &	Any type fulfilling a Geometry Concept	geometry2	A model of the specified concept

Returns

The calculated distance

Header

Either

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

Or

```
#include <boost/geometry/algorithms/distance.hpp>
```

Conformance

The function distance implements function Distance from the [OGC Simple Feature Specification](#).

Supported geometries

	Point	Segment	Box	Lines - tring	Ring	Polygon	Multi - Point	MultiLin- estring	MultiPoly- gon
Point	✓	✓	✗	✓	✓	✓	✓	✓	✓
Segment	✓	✗	✗	✗	✗	✗	✗	✗	✗
Box	✗	✗	✗	✗	✗	✗	✗	✗	✗
Linestring	✓	✗	✗	✗	✗	✗	✓	✗	✗
Ring	✓	✗	✗	✗	✗	✗	✗	✗	✗
Polygon	✓	✗	✗	✗	✗	✗	✓	✗	✗
Multi - Point	✓	✗	✗	✓	✗	✓	✓	✓	✓
MultiLin- estring	✓	✗	✗	✗	✗	✗	✓	✗	✗
MultiPoly- gon	✓	✗	✗	✗	✗	✗	✓	✗	✗

Complexity

Linear

For multi-geometry to multi-geometry: currently quadratic

Example

Shows calculation of distance of point to some other geometries

```

#include <iostream>
#include <list>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/linestring.hpp>
#include <boost/geometry/geometries/point_xy.hpp>
#include <boost/geometry/geometries/polygon.hpp>
#include <boost/geometry/multi/geometries/multi_point.hpp>
#include <boost/geometry/multi/geometries/multi_polygon.hpp>

#include <boost/geometry/io/wkt/wkt.hpp>

#include <boost/foreach.hpp>

int main()
{
    typedef boost::geometry::model::d2::point_xy<double> point_type;
    typedef boost::geometry::model::polygon<point_type> polygon_type;
    typedef boost::geometry::model::linestring<point_type> linestring_type;
    typedef boost::geometry::model::multi_point<point_type> multi_point_type;

    point_type p(1,2);
    polygon_type poly;
    linestring_type line;
    multi_point_type mp;

    boost::geometry::read_wkt(
        "POLYGON((2 1.3,2.4 1.7,2.8 1.8,3.4 1.2,3.7 1.6,3.4 2,4.1 3,5.3 2.6,5.4 1.2,4.9 0.8,2.9 ↵
0.7,2 1.3))"
        "(4.0 2.0, 4.2 1.4, 4.8 1.9, 4.4 2.2, 4.0 2.0))", poly);
    line.push_back(point_type(0,0));
    line.push_back(point_type(0,3));
    mp.push_back(point_type(0,0));
    mp.push_back(point_type(3,3));

    std::cout
        << "Point-Poly: " << boost::geometry::distance(p, poly) << std::endl
        << "Point-Line: " << boost::geometry::distance(p, line) << std::endl
        << "Point-MultiPoint: " << boost::geometry::distance(p, mp) << std::endl;

    return 0;
}

```

Output:

```

Point-Poly: 1.22066
Point-Line: 1
Point-MultiPoint: 2.23607

```

distance (with strategy)

Calculate the distance of two geometries using the specified strategy.

Description

The free function `area` calculates the area of a geometry. using the specified strategy. Reasons to specify a strategy include: use another coordinate system for calculations; construct the strategy beforehand (e.g. with the radius of the Earth); select a strategy when there are more than one available for a calculation.

Synopsis

```
template<typename Geometry1, typename Geometry2, typename Strategy>
strategy::distance::services::return_type<Strategy>::type distance(Geometry1 const & geometry1, Geometry2 const & geometry2, Strategy const & strategy)
```

Parameters

Type	Concept	Name	Description
Geometry1 const &	Any type fulfilling a Geometry Concept	geometry1	A model of the specified concept
Geometry2 const &	Any type fulfilling a Geometry Concept	geometry2	A model of the specified concept
Strategy const &	Any type fulfilling a Distance Strategy Concept	strategy	The strategy which will be used for distance calculations

Returns

The calculated distance

Header

Either

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

Or

```
#include <boost/geometry/algorithms/distance.hpp>
```

Available Strategies

- [Pythagoras \(cartesian\)](#)
- [Haversine \(spherical\)](#)
- [Cross track \(spherical, point-to-segment\)](#)
- [Projected point \(cartesian, point-to-segment\)](#)
- more (currently extensions): [Vincenty](#), [Andoyer](#) (geographic)

envelope

envelope

Calculates the envelope of a geometry.

Description

The free function `envelope` calculates the envelope (also known as axis aligned bounding box, aabb, or minimum bounding rectangle, mbr) of a geometry.

Synopsis

```
template<typename Geometry, typename Box>
void envelope(Geometry const & geometry, Box & mbr)
```

Parameters

Type	Concept	Name	Description
Geometry const &	Any type fulfilling a Geometry Concept	geometry	A model of the specified concept
Box &	Any type fulfilling a Box Concept	mbr	A model of the specified Box Concept which is set to the envelope

Header

Either

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

Or

```
#include <boost/geometry/algorithms/envelope.hpp>
```

Conformance

The function envelope implements function Envelope from the [OGC Simple Feature Specification](#).

Example

Shows how to calculate the bounding box of a polygon

```

#include <iostream>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/box.hpp>
#include <boost/geometry/geometries/point_xy.hpp>
#include <boost/geometry/geometries/polygon.hpp>
#include <boost/geometry/io/wkt/wkt.hpp>

int main()
{
    typedef boost::geometry::model::d2::point_xy<double> point;

    boost::geometry::model::polygon<point> polygon;

    boost::geometry::read_wkt(
        "POLYGON((2 1.3,2.4 1.7,2.8 1.8,3.4 1.2,3.7 1.6,3.4 2,4.1 3,5.3 2.6,5.4 1.2,4.9 0.8,2.9 ↵
0.7,2 1.3))"
        "(4.0 2.0, 4.2 1.4, 4.8 1.9, 4.4 2.2, 4.0 2.0))", polygon);

    boost::geometry::model::box<point> box;
    boost::geometry::envelope(polygon, box);

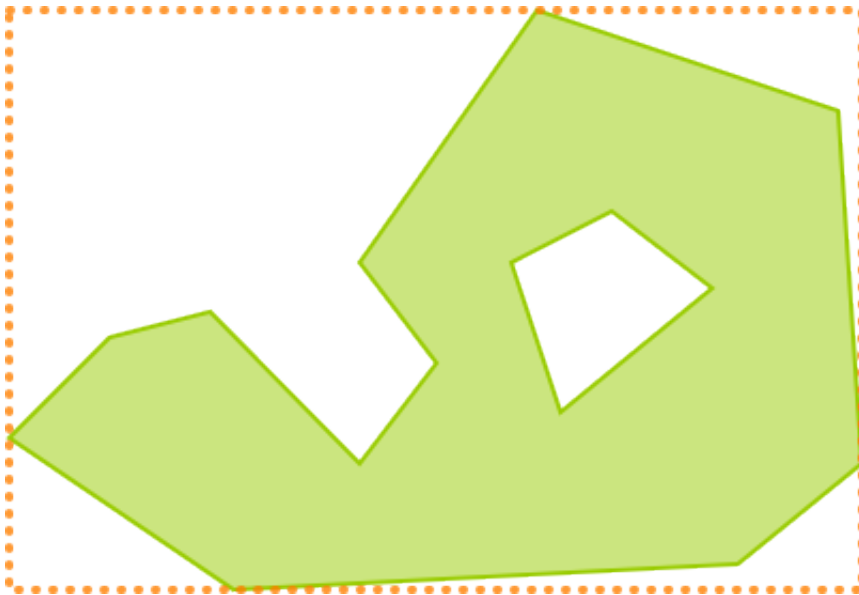
    std::cout << "envelope:" << boost::geometry::dsv(box) << std::endl;

    return 0;
}

```

Output:

```
envelope:((2, 0.7), (5.4, 3))
```



return_envelope

Calculates the envelope of a geometry.

Description

The free function `return_envelope` calculates the envelope (also known as axis aligned bounding box, aabb, or minimum bounding rectangle, mbr) of a geometry. This version with the `return_` prefix returns the envelope, and a template parameter must therefore be specified in the call.

Synopsis

```
template<typename Box, typename Geometry>
Box return_envelope(Geometry const & geometry)
```

Parameters

Type	Concept	Name	Description
Box	Any type fulfilling a Box Concept	-	Must be specified
Geometry const &	Any type fulfilling a Geometry Concept	geometry	A model of the specified concept

Returns

The calculated envelope

Header

Either

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

Or

```
#include <boost/geometry/algorithms/envelope.hpp>
```

Conformance

The function `envelope` implements function `Envelope` from the [OGC Simple Feature Specification](#).

Example

Shows how to return the envelope of a ring

```
#include <iostream>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/box.hpp>
#include <boost/geometry/geometries/point_xy.hpp>
#include <boost/geometry/geometries/ring.hpp>

#include <boost/assign.hpp>

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

    typedef boost::geometry::model::d2::point_xy<double> point;

    boost::geometry::model::ring<point> ring;
    ring +=
        point(4.0, -0.5), point(3.5, 1.0),
        point(2.0, 1.5), point(3.5, 2.0),
        point(4.0, 3.5), point(4.5, 2.0),
        point(6.0, 1.5), point(4.5, 1.0),
        point(4.0, -0.5);

    typedef boost::geometry::model::box<point> box;

    std::cout
        << "return_envelope:"
        << boost::geometry::dsv(boost::geometry::return_envelope<box>(ring))
        << std::endl;

    return 0;
}
```

Output:

```
return_envelope:((2, -0.5), (6, 3.5))
```



equals

Checks if a geometry are spatially equal.

Description

The free function equals checks if the first geometry is spatially equal the second geometry. Spatially equal means that the same point set is included. A box can therefore be spatially equal to a ring or a polygon, or a linestring can be spatially equal to a multi-linestring or a segment. This only theoretically, not all combinations are implemented yet.

Synopsis

```
template<typename Geometry1, typename Geometry2>
bool equals(Geometry1 const & geometry1, Geometry2 const & geometry2)
```

Parameters

Type	Concept	Name	Description
Geometry1 const &	Any type fulfilling a Geometry Concept	geometry1	A model of the specified concept
Geometry2 const &	Any type fulfilling a Geometry Concept	geometry2	A model of the specified concept

Returns

Returns true if two geometries are spatially equal

Header

Either

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

Or

```
#include <boost/geometry/algorithms/equals.hpp>
```

Conformance

The function equals implements function Equals from the [OGC Simple Feature Specification](#).

Supported geometries

	Point	Box	Linestring	Ring	Polygon	MultiPolygon
Point						
Box						
Linestring						
Ring						
Polygon						
MultiPolygon						

Complexity

Linear

Example

Shows the predicate equals, which returns true if two geometries are spatially equal

```

#include <iostream>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/polygon.hpp>
#include <boost/geometry/geometries/adapted/boost_tuple.hpp>

BOOST_GEOMETRY_REGISTER_BOOST_TUPLE_CS(cs::cartesian)

#include <boost/assign.hpp>

int main()
{
    using boost::assign::tuple_list_of;

    typedef boost::tuple<int, int> point;

    boost::geometry::model::polygon<point> poly1, poly2;
    boost::geometry::exterior_ring(poly1) = tuple_list_of(0, 0)(0, 5)(5, 5)(5, 0)(0, 0);
    boost::geometry::exterior_ring(poly2) = tuple_list_of(5, 0)(0, 0)(0, 5)(5, 5)(5, 0);

    std::cout
        << "polygons are spatially "
        << (boost::geometry::equals(poly1, poly2) ? "equal" : "not equal")
        << std::endl;

    boost::geometry::model::box<point> box;
    boost::geometry::assign_values(box, 0, 0, 5, 5);

    std::cout
        << "polygon and box are spatially "
        << (boost::geometry::equals(box, poly2) ? "equal" : "not equal")
        << std::endl;

    return 0;
}

```

Output:

```

polygons are spatially equal
polygon and box are spatially equal

```

expand

Expands a box using the bounding box (envelope) of another geometry (box, point)

Synopsis

```

template<typename Box, typename Geometry>
void expand(Box & box, Geometry const & geometry)

```


Parameters

Type	Concept	Name	Description
Box &	type of the box	box	box to be expanded using another geometry, mutable
Geometry const &	Any type fulfilling a Geometry Concept	geometry	A model of the specified concept geometry which envelope (bounding box) will be added to the box

Header

Either

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

Or

```
#include <boost/geometry/algorithms/expand.hpp>
```

Conformance

The function `expand` is not defined by OGC.

Behavior

Case	Behavior
Rectangle / Point	Box is expanded to include the specified Point
Rectangle / Rectangle	Box is expanded to include the specified Rectangle
Rectangle / Other geometries	Not yet supported in this version



Note

To use `expand` with another geometry type then specified, use `expand(make_envelope<box_type>(geometry))`

Complexity

Linear

Example

Shows the usage of `expand`

```

#include <iostream>
#include <list>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/box.hpp>
#include <boost/geometry/geometries/point_xy.hpp>

int main()
{
    typedef boost::geometry::model::d2::point_xy<short int> point_type;
    typedef boost::geometry::model::box<point_type> box_type;

    using boost::geometry::expand;

    box_type box = boost::geometry::make_inverse<box_type>(); ❶

    expand(box, point_type(0, 0));
    expand(box, point_type(1, 2));
    expand(box, point_type(5, 4));
    expand(box, boost::geometry::make<box_type>(3, 3, 5, 5));

    std::cout << boost::geometry::dsv(box) << std::endl;

    return 0;
}

```

❶ `expand` is usually preceded by a call to `assign_inverse` or `make_inverse`

Output:

```
((0, 0), (5, 5))
```

for_each

for_each_point (const version)

Applies function **f** to each point.

Description

Applies a function **f** (functor, having `operator()` defined) to each point making up the geometry

Synopsis

```

template<typename Geometry, typename Functor>
Functor for_each_point(Geometry const & geometry, Functor f)

```

Parameters

Type	Concept	Name	Description
Geometry const &	Any type fulfilling a Geometry Concept	geometry	A model of the specified concept
Functor	Function or class with <code>operator()</code>	f	Unary function, taking a const point as argument

Header

Either

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

Or

```
#include <boost/geometry/algorithms/for_each.hpp>
```

Conformance

The function `for_each_point` is not defined by OGC.

The function `for_each_point` conforms to the `std::for_each` function of the C++ std-library.

Example

Sample using `for_each_point`, using a function to list coordinates

```
#include <iostream>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/point_xy.hpp>
#include <boost/geometry/geometries/polygon.hpp>
#include <boost/geometry/io/wkt/wkt.hpp>

template <typename Point>
void list_coordinates(Point const& p)
{
    using boost::geometry::get;
    std::cout << "x = " << get<0>(p) << " y = " << get<1>(p) << std::endl;
}

int main()
{
    typedef boost::geometry::model::d2::point_xy<double> point;
    boost::geometry::model::polygon<point> poly;
    boost::geometry::read_wkt("POLYGON((0 0,0 4,4 0,0 0))", poly);
    boost::geometry::for_each_point(poly, list_coordinates<point>);
    return 0;
}
```

Output:

```
x = 0 y = 0
x = 0 y = 4
x = 4 y = 0
x = 0 y = 0
```

for_each_point

Applies function **f** to each point.

Description

Applies a function **f** (functor, having operator() defined) to each point making up the geometry

Synopsis

```
template<typename Geometry, typename Functor>
Functor for_each_point(Geometry & geometry, Functor f)
```

Parameters

Type	Concept	Name	Description
Geometry &	Any type fulfilling a Geometry Concept	geometry	A model of the specified concept
Functor	Function or class with operator()	f	Unary function, taking a point as argument

Header

Either

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

Or

```
#include <boost/geometry/algorithms/for_each.hpp>
```

Conformance

The function `for_each_point` is not defined by OGC.

The function `for_each_point` conforms to the `std::for_each` function of the C++ std-library.

Example

Convenient usage of `for_each_point`, rounding all points of a geometry

```

#include <iostream>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/point_xy.hpp>
#include <boost/geometry/geometries/polygon.hpp>
#include <boost/geometry/io/wkt/wkt.hpp>

template <typename Point>
class round_coordinates
{
private :
    typedef typename boost::geometry::coordinate_type<Point>::type coordinate_type;
    coordinate_type factor;

    inline coordinate_type round(coordinate_type value)
    {
        return floor(0.5 + (value / factor)) * factor;
    }

public :
    round_coordinates(coordinate_type f)
        : factor(f)
    {}

    inline void operator()(Point& p)
    {
        using boost::geometry::get;
        using boost::geometry::set;
        set<0>(p, round(get<0>(p)));
        set<1>(p, round(get<1>(p)));
    }
};

int main()
{
    typedef boost::geometry::model::d2::point_xy<double> point;
    boost::geometry::model::polygon<point> poly;
    boost::geometry::read_wkt("POLYGON((0 0,1.123 9.987,8.876 2.234,0 0),(3.345 4.456,7.654 ↵
8.765,9.123 5.432,3.345 4.456))", poly);
    boost::geometry::for_each_point(poly, round_coordinates<point>(0.1));
    std::cout << "Rounded: " << boost::geometry::wkt(poly) << std::endl;
    return 0;
}

```

Output:

```
Rounded: POLYGON((0 0,1.1 10,8.9 2.2,0 0),(3.3 4.5,7.7 8.8,9.1 5.4,3.3 4.5))
```

for_each_segment (const version)

Applies function **f** to each segment.

Description

Applies a function **f** (functor, having operator() defined) to each segment making up the geometry

Synopsis

```
template<typename Geometry, typename Functor>
Functor for_each_segment(Geometry const & geometry, Functor f)
```

Parameters

Type	Concept	Name	Description
Geometry const &	Any type fulfilling a Geometry Concept	geometry	A model of the specified concept
Functor	Function or class with operator()	f	Unary function, taking a const segment as argument

Header

Either

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

Or

```
#include <boost/geometry/algorithms/for_each.hpp>
```

Conformance

The function `for_each_segment` is not defined by OGC.

The function `for_each_segment` conforms to the `std::for_each` function of the C++ std-library.

Example

Sample using `for_each_segment`, using a functor to get the minimum and maximum length of a segment in a linestring

```

#include <iostream>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/linestring.hpp>
#include <boost/geometry/geometries/point_xy.hpp>

#include <boost/assign.hpp>

template <typename Segment>
struct gather_segment_statistics
{
    // Remember that if coordinates are integer, the length might be floating point
    // So use "double" for integers. In other cases, use coordinate type
    typedef typename boost::geometry::select_most_precise
    <
        typename boost::geometry::coordinate_type<Segment>::type,
        double
    >::type type;

    type min_length, max_length;

    // Initialize min and max
    gather_segment_statistics()
        : min_length(1e38)
        , max_length(-1)
    {}

    // This operator is called for each segment
    inline void operator()(Segment const& s)
    {
        type length = boost::geometry::length(s);
        if (length < min_length) min_length = length;
        if (length > max_length) max_length = length;
    }
};

int main()
{
    // Bring "+=" for a vector into scope
    using namespace boost::assign;

    // Define a type
    typedef boost::geometry::model::d2::point_xy<double> point;

    // Declare a linestring
    boost::geometry::model::linestring<point> polyline;

    // Use Boost.Assign to initialize a linestring
    polyline += point(0, 0), point(3, 3), point(5, 1), point(6, 2),
        point(8, 0), point(4, -4), point(1, -1), point(3, 2);

    // Declare the gathering class...
    gather_segment_statistics
    <
        boost::geometry::model::referring_segment<point>
    > functor;

    // ... and use it, the essence.
    // As also in std::for_each it is a const value, so retrieve it as a return value.
    functor = boost::geometry::for_each_segment(polyline, functor);
}

```

```
// Output the results
std::cout
    << "Min segment length: " << functor.min_length << std::endl
    << "Max segment length: " << functor.max_length << std::endl;

return 0;
}
```

Output:

```
Min segment length: 1.41421
Max segment length: 5.65685
```

for_each_segment

Applies function **f** to each segment.

Description

Applies a function **f** (functor, having operator() defined) to each segment making up the geometry

Synopsis

```
template<typename Geometry, typename Functor>
Functor for_each_segment(Geometry & geometry, Functor f)
```

Parameters

Type	Concept	Name	Description
Geometry &	Any type fulfilling a Geometry Concept	geometry	A model of the specified concept
Functor	Function or class with operator()	f	Unary function, taking a segment as argument

Header

Either

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

Or

```
#include <boost/geometry/algorithms/for_each.hpp>
```

Conformance

The function `for_each_segment` is not defined by OGC.

The function `for_each_segment` conforms to the `std::for_each` function of the C++ std-library.

intersection

Calculate the intersection of two geometries.

Description

The free function intersection calculates the spatial set theoretic intersection of two geometries.

Synopsis

```
template<typename Geometry1, typename Geometry2, typename GeometryOut>
bool intersection(Geometry1 const & geometry1, Geometry2 const & geometry2, GeometryOut & geometry_out)
```

Parameters

Type	Concept	Name	Description
Geometry1 const &	Any type fulfilling a Geometry Concept	geometry1	A model of the specified concept
Geometry2 const &	Any type fulfilling a Geometry Concept	geometry2	A model of the specified concept
GeometryOut &	Collection of geometries (e.g. std::vector, std::deque, boost::geometry::multi*) of which the value_type fulfills a Point, LineString or Polygon concept, or it is the output geometry (e.g. for a box)	geometry_out	The output geometry, either a multi_point, multi_polygon, multi_linestring, or a box (for intersection of two boxes)

Header

Either

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

Or

```
#include <boost/geometry/algorithms/intersection.hpp>
```

Conformance

The function intersection implements function Intersection from the [OGC Simple Feature Specification](#).

Behavior

Case	Behavior
GeometryOut is a Point	Calculates intersection points of input geometries
GeometryOut is a Linestring	Calculates intersection linestrings of input geometries (NYI)
GeometryOut is a Polygon	Calculates intersection polygons of input (multi)polygons and/or boxes



Note

Check the [Polygon Concept](#) for the rules that polygon input for this algorithm should fulfill

Example

Shows the intersection of two polygons

```
#include <iostream>
#include <deque>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/point_xy.hpp>
#include <boost/geometry/geometries/polygon.hpp>
#include <boost/geometry/io/wkt/wkt.hpp>

#include <boost/foreach.hpp>

int main()
{
    typedef boost::geometry::model::polygon<boost::geometry::model::d2::point_xy<double> > polygon;

    polygon green, blue;

    boost::geometry::read_wkt(
        "POLYGON((2 1.3,2.4 1.7,2.8 1.8,3.4 1.2,3.7 1.6,3.4 2,4.1 3,5.3 2.6,5.4 1.2,4.9 0.8,2.9 ↵
0.7,2 1.3))"
        "(4.0 2.0, 4.2 1.4, 4.8 1.9, 4.4 2.2, 4.0 2.0))", green);

    boost::geometry::read_wkt(
        "POLYGON((4.0 -0.5 , 3.5 1.0 , 2.0 1.5 , 3.5 2.0 , 4.0 3.5 , 4.5 2.0 , 6.0 1.5 , 4.5 ↵
1.0 , 4.0 -0.5))", blue);

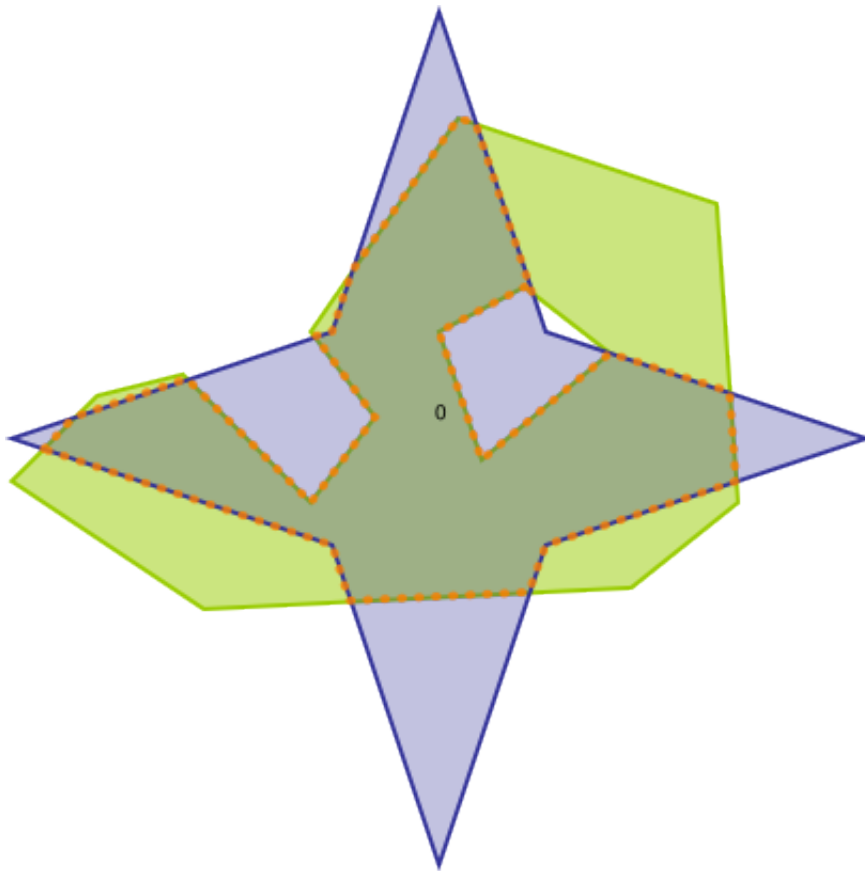
    std::deque<polygon> output;
    boost::geometry::intersection(green, blue, output);

    int i = 0;
    std::cout << "green && blue:" << std::endl;
    BOOST_FOREACH(polygon const& p, output)
    {
        std::cout << i++ << ": " << boost::geometry::area(p) << std::endl;
    }

    return 0;
}
```

Output:

```
green && blue:  
0: 2.50205
```



See also

- [union](#)
- [difference](#)
- [sym_difference](#) (symmetric difference)

intersects

intersects (one geometry)

Checks if a geometry has at least one intersection (crossing or self-tangency)

Synopsis

```
template<typename Geometry>  
bool intersects(Geometry const & geometry)
```

Parameters

Type	Concept	Name	Description
Geometry const &	Any type fulfilling a Geometry Concept	geometry	A model of the specified concept

Returns

Returns true if the geometry is self-intersecting

Header

Either

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

Or

```
#include <boost/geometry/algorithms/intersects.hpp>
```

Conformance

The function intersects implements function Intersects from the [OGC Simple Feature Specification](#).

The version with one parameter is additional and not described in the OGC standard

Examples

Check if two linestrings intersect each other

```
#include <iostream>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/linestring.hpp>
#include <boost/geometry/geometries/point_xy.hpp>
#include <boost/geometry/io/wkt/wkt.hpp>

int main()
{
    // Calculate the intersects of a cartesian polygon
    typedef boost::geometry::model::d2::point_xy<double> P;
    boost::geometry::model::linestring<P> line1, line2;

    boost::geometry::read_wkt("linestring(1 1,2 2,3 3)", line1);
    boost::geometry::read_wkt("linestring(2 1,1 2,4 0)", line2);

    bool b = boost::geometry::intersects(line1, line2);

    std::cout << "Intersects: " << (b ? "YES" : "NO") << std::endl;

    return 0;
}
```

Output:

```
Intersects: YES
```

intersects (two geometries)

Checks if two geometries have at least one intersection.

Synopsis

```
template<typename Geometry1, typename Geometry2>
bool intersects(Geometry1 const & geometry1, Geometry2 const & geometry2)
```

Parameters

Type	Concept	Name	Description
Geometry1 const &	Any type fulfilling a Geometry Concept	geometry1	A model of the specified concept
Geometry2 const &	Any type fulfilling a Geometry Concept	geometry2	A model of the specified concept

Returns

Returns true if two geometries intersect each other

Header

Either

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

Or

```
#include <boost/geometry/algorithms/intersects.hpp>
```

Conformance

The function intersects implements function Intersects from the [OGC Simple Feature Specification](#).

The version with one parameter is additional and not described in the OGC standard

Examples

Check if two linestrings intersect each other

```
#include <iostream>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/linestring.hpp>
#include <boost/geometry/geometries/point_xy.hpp>
#include <boost/geometry/io/wkt/wkt.hpp>

int main()
{
    // Calculate the intersects of a cartesian polygon
    typedef boost::geometry::model::d2::point_xy<double> P;
    boost::geometry::model::linestring<P> line1, line2;

    boost::geometry::read_wkt("linestring(1 1,2 2,3 3)", line1);
    boost::geometry::read_wkt("linestring(2 1,1 2,4 0)", line2);

    bool b = boost::geometry::intersects(line1, line2);

    std::cout << "Intersects: " << (b ? "YES" : "NO") << std::endl;

    return 0;
}
```

Output:

```
Intersects: YES
```

length

length

Calculates the length of a geometry.

Description

The free function `length` calculates the length (the sum of distances between consecutive points) of a geometry. It uses the default strategy, based on the coordinate system of the geometry.

Synopsis

```
template<typename Geometry>
default_length_result<Geometry>::type length(Geometry const & geometry)
```

Parameters

Type	Concept	Name	Description
Geometry const &	Any type fulfilling a Geometry Concept	geometry	A model of the specified concept

Returns

The calculated length

Header

Either

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

Or

```
#include <boost/geometry/algorithms/length.hpp>
```

Conformance

The function `length` implements function `Length` from the [OGC Simple Feature Specification](#).

Behavior

Case	Behavior
pointlike (e.g. <code>point</code>)	Returns 0
linear (e.g. <code>linestring</code>)	Returns the length
areal (e.g. <code>polygon</code>)	Returns 0

Complexity

Linear

Examples

The following simple example shows the calculation of the length of a `linestring` containing three points

```
#include <iostream>
#include <boost/geometry.hpp>
#include <boost/geometry/geometries/linestring.hpp>
#include <boost/geometry/geometries/point_xy.hpp>
#include <boost/geometry/io/wkt/wkt.hpp>

int main()
{
    using namespace boost::geometry;
    model::linestring<model::d2::point_xy<double> > line;
    read_wkt("linestring(0 0,1 1,4 8,3 2)", line);
    std::cout << "linestring length is "
                << length(line)
                << " units" << std::endl;

    return 0;
}
```

Output:

```
linestring length is 15.1127 units
```

length (with strategy)

Calculates the length of a geometry using the specified strategy.

Description

The free function `length` calculates the length (the sum of distances between consecutive points) of a geometry using the specified strategy. Reasons to specify a strategy include: use another coordinate system for calculations; construct the strategy beforehand (e.g. with the radius of the Earth); select a strategy when there are more than one available for a calculation.

Synopsis

```
template<typename Geometry, typename Strategy>
default_length_result<Geometry>::type length(Geometry const & geometry, Strategy const & strategy)
```

Parameters

Type	Concept	Name	Description
Geometry const &	Any type fulfilling a Geometry Concept	geometry	A model of the specified concept
Strategy const &	Any type fulfilling a distance Strategy Concept	strategy	The strategy which will be used for distance calculations

Returns

The calculated length

Header

Either

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

Or

```
#include <boost/geometry/algorithms/length.hpp>
```

Conformance

The function `length` implements function `Length` from the [OGC Simple Feature Specification](#).

Behavior

Case	Behavior
pointlike (e.g. point)	Returns 0
linear (e.g. linestring)	Returns the length
areal (e.g. polygon)	Returns 0

Complexity

Linear

Examples

The following example shows the length measured over a sphere, expressed in kilometers. To do that the radius of the sphere must be specified in the constructor of the strategy.


```

#include <iostream>
#include <boost/geometry.hpp>
#include <boost/geometry/geometries/linestring.hpp>

int main()
{
    using namespace boost::geometry;
    typedef model::point<float, 2, cs::spherical_equatorial<degree> > P;
    model::linestring<P> line;
    line.push_back(P(2, 41));
    line.push_back(P(2, 48));
    line.push_back(P(5, 52));
    double const mean_radius = 6371.0; ❶
    std::cout << "length is "
        << length(line, strategy::distance::haversine<P>(mean_radius) )
        << " kilometers " << std::endl;

    return 0;
}

```

❶ [Wiki](#)

Output:

```
length is 1272.03 kilometers
```

make

make (2 coordinate values)

Construct a geometry.

Synopsis

```

template<typename Geometry, typename Type>
Geometry make(Type const & c1, Type const & c2)

```

Parameters

Type	Concept	Name	Description
Geometry	Any type fulfilling a Geometry Concept	-	Must be specified
Type const &	numerical type (int, double, ttmath, ...) to specify the co-ordinates	c1	First coordinate (usually x-coordinate)
Type const &	numerical type (int, double, ttmath, ...) to specify the co-ordinates	c2	Second coordinate (usually y-coordinate)

Returns

The constructed geometry, here: a 2D point

Header

Either

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

Or

```
#include <boost/geometry/algorithms/make.hpp>
```

Example

Shows the usage of make as a generic constructor for different point types

```
#include <iostream>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/point_xy.hpp>
#include <boost/geometry/geometries/register/point.hpp>
#include <boost/geometry/geometries/adapted/boost_tuple.hpp>
#include <boost/geometry/geometries/adapted/boost_polygon/point.hpp>

BOOST_GEOMETRY_REGISTER_BOOST_TUPLE_CS(cs::cartesian)

struct mypoint { float _x, _y; };

BOOST_GEOMETRY_REGISTER_POINT_2D(mypoint, float, cs::cartesian, _x, _y)

template <typename Point>
void construct_and_display()
{
    using boost::geometry::make;
    using boost::geometry::get;

    Point p = make<Point>(1, 2);

    std::cout << "x=" << get<0>(p) << " y=" << get<1>(p)
        << " (" << typeid(Point).name() << ")"
        << std::endl;
}

int main()
{
    construct_and_display<boost::geometry::model::d2::point_xy<double>>>();
    construct_and_display<boost::geometry::model::d2::point_xy<int>>>();
    construct_and_display<boost::tuple<double, double>>>();
    construct_and_display<boost::polygon::point_data<int>>>();
    construct_and_display<mypoint>>();
    return 0;
}
```

Output (compiled using gcc):

```
x=1 y=2 (N5boost8geometry5model2d28point_xyIdNS0_2cs9cartesianEEE)
x=1 y=2 (N5boost8geometry5model2d28point_xyIiNS0_2cs9cartesianEEE)
x=1 y=2 (N5boost6tuples5tupleIdNS0_9null_typeES2_S2_S2_S2_S2_S2_S2_EE)
x=1 y=2 (N5boost7polygon10point_dataIiEE)
x=1 y=2 (7mypoint)
```

See also

- [assign](#)

make (3 coordinate values)

Construct a geometry.

Synopsis

```
template<typename Geometry, typename Type>
Geometry make(Type const & c1, Type const & c2, Type const & c3)
```

Parameters

Type	Concept	Name	Description
Geometry	Any type fulfilling a Geometry Concept	-	Must be specified
Type const &	numerical type (int, double, ttmath, ...) to specify the co-ordinates	c1	First coordinate (usually x-coordinate)
Type const &	numerical type (int, double, ttmath, ...) to specify the co-ordinates	c2	Second coordinate (usually y-coordinate)
Type const &	numerical type (int, double, ttmath, ...) to specify the co-ordinates	c3	Third coordinate (usually z-coordinate)

Returns

The constructed geometry, here: a 3D point

Header

Either

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

Or

```
#include <boost/geometry/algorithms/make.hpp>
```

Example

Using make to construct a three dimensional point

```
#include <iostream>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/point.hpp>

int main()
{
    typedef boost::geometry::model::point<double, 3, boost::geometry::cs::cartesian> point_type;
    point_type p = boost::geometry::make<point_type>(1, 2, 3);
    std::cout << boost::geometry::dsv(p) << std::endl;
    return 0;
}
```

Output:

```
( 1, 2, 3 )
```

See also

- [assign](#)

make_inverse

Construct a box with inverse infinite coordinates.

Description

The `make_inverse` function initializes a 2D or 3D box with large coordinates, the min corner is very large, the max corner is very small. This is useful e.g. in combination with the `expand` function, to determine the bounding box of a series of geometries.

Synopsis

```
template<typename Geometry>
Geometry make_inverse()
```

Parameters

Type	Concept	Name	Description
Geometry	Any type fulfilling a Geometry Concept	-	Must be specified

Returns

The constructed geometry, here: a box

Header

Either

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

Or

```
#include <boost/geometry/algorithms/make.hpp>
```

Example

Usage of `make_inverse` and `expand` to conveniently determine bounding box of several objects

```

#include <iostream>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/box.hpp>
#include <boost/geometry/geometries/point_xy.hpp>

using namespace boost::geometry;

int main()
{
    typedef model::d2::point_xy<double> point;
    typedef model::box<point> box;

    box all = make_inverse<box>();
    std::cout << dsv(all) << std::endl;
    expand(all, make<box>(0, 0, 3, 4));
    expand(all, make<box>(2, 2, 5, 6));
    std::cout << dsv(all) << std::endl;

    return 0;
}

```

Output:

```

((1.79769e+308, 1.79769e+308), (-1.79769e+308, -1.79769e+308))
((0, 0), (5, 6))

```

See also

- [assign_inverse](#)

make_zero

Construct a geometry with its coordinates initialized to zero.

Description

The `make_zero` function initializes a 2D or 3D point or box with coordinates of zero

Synopsis

```

template<typename Geometry>
Geometry make_zero()

```

Parameters

Type	Concept	Name	Description
Geometry	Any type fulfilling a Geometry Concept	-	Must be specified

Returns

The constructed and zero-initialized geometry

Header

Either

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

Or

```
#include <boost/geometry/algorithms/make.hpp>
```

num_geometries

Calculates the number of geometries of a geometry.

Description

The free function `num_geometries` calculates the number of geometries of a geometry.

Synopsis

```
template<typename Geometry>
std::size_t num_geometries(Geometry const & geometry)
```

Parameters

Type	Concept	Name	Description
Geometry const &	Any type fulfilling a Geometry Concept	geometry	A model of the specified concept

Returns

The calculated number of geometries

Header

Either

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

Or

```
#include <boost/geometry/algorithms/num_geometries.hpp>
```

Conformance

The function `num_geometries` implements function `NumGeometries` from the [OGC Simple Feature Specification](#).

Behavior

Case	Behavior
single (e.g. point, polygon)	Returns 1
multiple (e.g. multi_point, multi_polygon)	Returns <code>boost::size(geometry)</code> ; the input is considered as a range

Complexity

Constant

Examples

Get the number of geometries making up a multi-geometry

```

#include <iostream>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/point_xy.hpp>
#include <boost/geometry/geometries/polygon.hpp>
#include <boost/geometry/multi/geometries/multi_polygon.hpp>
#include <boost/geometry/io/wkt/wkt.hpp>
#include <boost/geometry/multi/io/wkt/wkt.hpp>

int main()
{
    boost::geometry::model::multi_polygon
        <
            boost::geometry::model::polygon
                <
                    boost::geometry::model::d2::point_xy<double>
                >
            > mp;
    boost::geometry::read_wkt("MULTIPOLYGON(((0 0,0 10,10 0,0 0),(1 1,1 9,9 1,1 1)),((10 10,10 7,7 10,10 10)))", mp);
    std::cout << "Number of geometries: " << boost::geometry::num_geometries(mp) << std::endl;

    return 0;
}

```

Output:

```
Number of geometries: 2
```

num_interior_rings

Calculates the number of interior rings of a geometry.

Description

The free function `num_interior_rings` calculates the number of interior rings of a geometry.

Synopsis

```

template<typename Geometry>
std::size_t num_interior_rings(Geometry const & geometry)

```

Parameters

Type	Concept	Name	Description
Geometry const &	Any type fulfilling a Geometry Concept	geometry	A model of the specified concept

Returns

The calculated number of interior rings

Header

Either

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

Or

```
#include <boost/geometry/algorithms/num_interior_rings.hpp>
```

Conformance

The function `num_interior_ring` implements function `NumInteriorRing` from the [OGC Simple Feature Specification](#).



Note

Boost.Geometry adds an "s"

Behavior

Case	Behavior
Polygon	Returns the number of its interior rings
Multi Polygon	Returns the number of the interior rings of all polygons
Other geometries	Returns 0

Complexity

Constant

Examples

Get the number of interior rings in a multi-polygon

```
#include <iostream>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/point_xy.hpp>
#include <boost/geometry/geometries/polygon.hpp>
#include <boost/geometry/multi/geometries/multi_polygon.hpp>
#include <boost/geometry/io/wkt/wkt.hpp>
#include <boost/geometry/multi/io/wkt/wkt.hpp>

int main()
{
    boost::geometry::model::multi_polygon
        <
            boost::geometry::model::polygon
                <
                    boost::geometry::model::d2::point_xy<double>
                >
            > mp;
    boost::geometry::read_wkt("MULTIPOLYGON(((0 0,0 10,10 0,0 0),(1 1,1 9,9 1,1 1)),((10 10,10 ↵
7,7 10,10 10)))", mp);
    std::cout << "Number of interior rings: " << boost::geometry::num_interi↵
or_rings(mp) << std::endl;

    return 0;
}
```

Output:

Number of interior rings: 1

num_points

Calculates the number of points of a geometry.

Description

The free function num_points calculates the number of points of a geometry.

Synopsis

```
template<typename Geometry>
std::size_t num_points(Geometry const & geometry, bool add_for_open = false)
```

Parameters

Type	Concept	Name	Description
Geometry const &	Any type fulfilling a Geometry Concept	geometry	A model of the specified concept
bool		add_for_open	add one for open geometries (i.e. polygon types which are not closed)

Returns

The calculated number of points

Header

Either

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

Or

```
#include <boost/geometry/algorithms/num_points.hpp>
```

Conformance

The function num_points implements function NumPoints from the [OGC Simple Feature Specification](#).



Note

num_points can be called for any geometry and not just linestrings (as the standard describes)

Behavior

Case	Behavior
Point	Returns 1
Segment	Returns 2
Rectangle	Returns 4
Rangelike (linestring, ring)	Returns boost::size(geometry)
Other geometries	Returns the sum of the number of points of its elements
Open geometries	Returns the sum of the number of points of its elements, it adds one for open (per ring) if specified
Closed geometries	Returns the sum of the number of points of its elements

Complexity

Constant or Linear

Examples

Get the number of points in a geometry

```
#include <iostream>

#include <boost/geometry.hpp>
#include <boost/geometry/multi/multi.hpp>
#include <boost/geometry/geometries/point_xy.hpp>
#include <boost/geometry/geometries/polygon.hpp>
#include <boost/geometry/multi/geometries/multi_polygon.hpp>
#include <boost/geometry/io/wkt/wkt.hpp>
#include <boost/geometry/multi/io/wkt/wkt.hpp>

int main()
{
    boost::geometry::model::multi_polygon
        <
            boost::geometry::model::polygon
                <
                    boost::geometry::model::d2::point_xy<double>
                >
            > mp;
    boost::geometry::read_wkt("MULTIPOLYGON(((0 0,0 10,10 0,0 0),(1 1,1 9,9 1,1 1)),((10 10,10 7,7 10,10 10)))", mp);
    std::cout << "Number of points: " << boost::geometry::num_points(mp) << std::endl;

    return 0;
}
```

Output:

```
Number of points: 12
```

overlaps

Checks if two geometries overlap.

Synopsis

```
template<typename Geometry1, typename Geometry2>
bool overlaps(Geometry1 const & geometry1, Geometry2 const & geometry2)
```

Parameters

Type	Concept	Name	Description
Geometry1 const &		geometry1	
Geometry2 const &		geometry2	

Returns

Returns true if two geometries overlap

Header

Either

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

Or

```
#include <boost/geometry/algorithms/overlaps.hpp>
```

Conformance

The function overlaps implements function Overlaps from the [OGC Simple Feature Specification](#).

perimeter

perimeter

Calculates the perimeter of a geometry.

Description

The function perimeter returns the perimeter of a geometry, using the default distance-calculation-strategy

Synopsis

```
template<typename Geometry>
default_length_result<Geometry>::type perimeter(Geometry const & geometry)
```

Parameters

Type	Concept	Name	Description
Geometry const &	Any type fulfilling a Geometry Concept	geometry	A model of the specified concept

Returns

The calculated perimeter

Header

Either

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

Or

```
#include <boost/geometry/algorithms/perimeter.hpp>
```

Conformance

The function `perimeter` is not defined by OGC.



Note

PostGIS contains an algorithm with the same name and the same functionality. See the [PostGIS documentation](#).

Behavior

Case	Behavior
pointlike (e.g. point)	Returns zero
linear (e.g. linestring)	Returns zero
areal (e.g. polygon)	Returns the perimeter

Complexity

Linear

`perimeter (with strategy)`

Calculates the perimeter of a geometry using the specified strategy.

Description

The function `perimeter` returns the perimeter of a geometry, using specified strategy

Synopsis

```
template<typename Geometry, typename Strategy>
default_length_result<Geometry>::type perimeter(Geometry const & geometry,
Strategy const & strategy)
```

Parameters

Type	Concept	Name	Description
Geometry const &	Any type fulfilling a Geometry Concept	geometry	A model of the specified concept
Strategy const &	Any type fulfilling a distance Strategy Concept	strategy	strategy to be used for distance calculations.

Returns

The calculated perimeter

Header

Either

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

Or

```
#include <boost/geometry/algorithms/perimeter.hpp>
```

Conformance

The function perimeter is not defined by OGC.



Note

PostGIS contains an algorithm with the same name and the same functionality. See the [PostGIS documentation](#).

Behavior

Case	Behavior
pointlike (e.g. point)	Returns zero
linear (e.g. linestring)	Returns zero
areal (e.g. polygon)	Returns the perimeter

Complexity

Linear

reverse

Reverses the points within a geometry.

Description

Generic function to reverse a geometry. It resembles the `std::reverse` functionality, but it takes the geometry type into account. Only for a ring or for a linestring it is the same as the `std::reverse`.

Synopsis

```
template<typename Geometry>
void reverse(Geometry & geometry)
```

Parameters

Type	Concept	Name	Description
Geometry &	Any type fulfilling a Geometry Concept	geometry	A model of the specified concept which will be reversed

Header

Either

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

Or

```
#include <boost/geometry/algorithms/reverse.hpp>
```

Conformance

The function reverse is not defined by OGC.

The function reverse conforms to the std::reverse function of the C++ std-library.

Behavior

Case	Behavior
Point	Nothing happens, geometry is unchanged
Segment	Not yet supported in this version
Rectangle	Nothing happens, geometry is unchanged
Linestring	Reverses the Linestring
Ring	Reverses the Ring
Polygon	Reverses the exterior ring and all interior rings in the polygon
Multi Point	Nothing happens, geometry is unchanged
Multi Linestring	Reverses all contained linestrings individually
Multi Polygon	Reverses all contained polygons individually



Note

The reverse of a (multi)polygon or ring might make a valid geometry invalid because the (counter)clockwise orientation reverses.

Complexity

Linear

Example

Shows how to reverse a ring or polygon

```
#include <iostream>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/polygon.hpp>
#include <boost/geometry/geometries/ring.hpp>
#include <boost/geometry/geometries/adapted/boost_tuple.hpp>

BOOST_GEOMETRY_REGISTER_BOOST_TUPLE_CS(cs::cartesian)

#include <boost/assign.hpp>

int main()
{
    using boost::assign::tuple_list_of;

    typedef boost::tuple<int, int> point;
    typedef boost::geometry::model::polygon<point> polygon;
    typedef boost::geometry::model::ring<point> ring;

    polygon poly;
    boost::geometry::exterior_ring(poly) = tuple_list_of(0, 0)(0, 9)(10, 10)(0, 0);
    boost::geometry::interior_rings(poly).push_back(tuple_list_of(1, 2)(4, 6)(2, 8)(1, 2));

    double area_before = boost::geometry::area(poly);
    boost::geometry::reverse(poly);
    double area_after = boost::geometry::area(poly);
    std::cout << boost::geometry::dsv(poly) << std::endl;
    std::cout << area_before << " -> " << area_after << std::endl;

    ring r = tuple_list_of(0, 0)(0, 9)(8, 8)(0, 0);

    area_before = boost::geometry::area(r);
    boost::geometry::reverse(r);
    area_after = boost::geometry::area(r);
    std::cout << boost::geometry::dsv(r) << std::endl;
    std::cout << area_before << " -> " << area_after << std::endl;

    return 0;
}
```

Output:

```
((0, 0), (10, 10), (0, 9), (0, 0)), ((1, 2), (2, 8), (4, 6), (1, 2)))
38 -> -38
((0, 0), (8, 8), (0, 9), (0, 0))
36 -> -36
```

simplify

simplify (with strategy)

Simplify a geometry using a specified strategy.

Synopsis

```
template<typename Geometry, typename Distance, typename Strategy>
void simplify(Strategy const & strategy, Geometry const & geometry, Geometry & out, Distance const & max_distance)
```

Parameters

Type	Concept	Name	Description
Strategy const &	A type fulfilling a SimplifyStrategy concept	strategy	simplify strategy to be used for simplification, might include point-distance strategy
Geometry const &	Any type fulfilling a Geometry Concept	geometry	input geometry, to be simplified
Geometry &	Any type fulfilling a Geometry Concept	out	output geometry, simplified version of the input geometry
Distance const &	A numerical distance measure	max_distance	distance (in units of input coordinates) of a vertex to other segments to be removed

Header

Either

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

Or

```
#include <boost/geometry/algorithms/simplify.hpp>
```

simplify

Simplify a geometry.

Synopsis

```
template<typename Geometry, typename Distance>
void simplify(Geometry const & geometry, Geometry & out, Distance const & max_distance)
```

Parameters

Type	Concept	Name	Description
Geometry const &	Any type fulfilling a Geometry Concept	geometry	input geometry, to be simplified
Geometry &	Any type fulfilling a Geometry Concept	out	output geometry, simplified version of the input geometry
Distance const &	numerical type (int, double, ttmath, ...)	max_distance	distance (in units of input coordinates) of a vertex to other segments to be removed

Header

Either

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

Or

```
#include <boost/geometry/algorithms/simplify.hpp>
```

Conformance

The function `simplify` is not defined by OGC.



Note

PostGIS contains an algorithm with the same name and the same functionality. See the [PostGIS documentation](#).



Note

SQL Server contains an algorithm `Reduce()` with the same functionality. See the [MSDN documentation](#).

Behavior

Simplification is done using [Douglas-Peucker](#) (if the default strategy is used).



Note

Geometries might become invalid by using `simplify`. The simplification process might create self-intersections.

Examples

Example showing how to simplify a linestring

```

#include <iostream>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/linestring.hpp>
#include <boost/geometry/geometries/point_xy.hpp>

❶
#include <boost/assign.hpp>

using namespace boost::assign;

int main()
{
    typedef boost::geometry::model::d2::point_xy<double> xy;

    boost::geometry::model::linestring<xy> line;
    line += xy(1.1, 1.1), xy(2.5, 2.1), xy(3.1, 3.1), xy(4.9, 1.1), xy(3.1, 1.9); ❷

    // Simplify it, using distance of 0.5 units
    boost::geometry::model::linestring<xy> simplified;
    boost::geometry::simplify(line, simplified, 0.5);
    std::cout
        << " original: " << boost::geometry::dsv(line) << std::endl
        << "simplified: " << boost::geometry::dsv(simplified) << std::endl;

    return 0;
}

```

- ❶ For this example we use Boost.Assign to add points
- ❷ With Boost.Assign

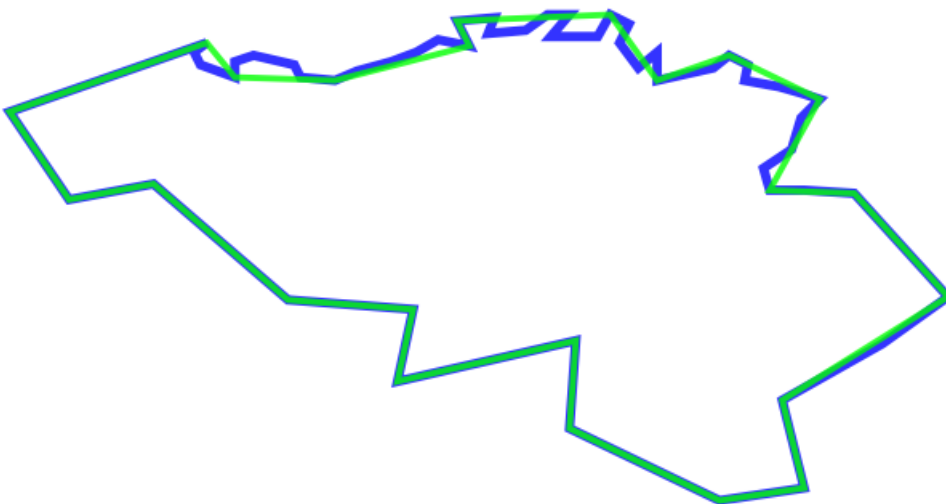
Output:

```

original: ((1.1, 1.1), (2.5, 2.1), (3.1, 3.1), (4.9, 1.1), (3.1, 1.9))
simplified: ((1.1, 1.1), (3.1, 3.1), (4.9, 1.1), (3.1, 1.9))

```

Image(s)



sym_difference

Calculate the symmetric difference of two geometries.

Description

The free function symmetric difference calculates the spatial set theoretic symmetric difference (XOR) of two geometries.

Synopsis

```
template<typename Geometry1, typename Geometry2, typename Collection>
void sym_difference(Geometry1 const & geometry1, Geometry2 const & geometry2, Collection & output_collection)
```

Parameters

Type	Concept	Name	Description
Geometry1 const &	Any type fulfilling a Geometry Concept	geometry1	A model of the specified concept
Geometry2 const &	Any type fulfilling a Geometry Concept	geometry2	A model of the specified concept
Collection &	output collection, either a multi-geometry, or a std::vector<Geometry> / std::deque<Geometry> etc	output_collection	the output collection

Header

Either

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

Or

```
#include <boost/geometry/algorithms/sym_difference.hpp>
```

Conformance

The function sym_difference implements function SymDifference from the [OGC Simple Feature Specification](http://www.opengeospatial.org/standards/feature).

Behavior

Case	Behavior
areal (e.g. polygon)	All combinations of: box, ring, polygon, multi_polygon
linear (e.g. linestring) / areal (e.g. polygon)	A combinations of a (multi) linestring with a (multi) polygon results in a collection of linestrings
Other geometries	Not yet supported in this version
Spherical	Not yet supported in this version
Three dimensional	Not yet supported in this version



Note

Check the [Polygon Concept](#) for the rules that polygon input for this algorithm should fulfill

Example

Shows how to calculate the symmetric difference (XOR) of two polygons

```
#include <iostream>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/polygon.hpp>
#include <boost/geometry/geometries/point_xy.hpp>
#include <boost/geometry/multi/geometries/multi_polygon.hpp>
#include <boost/geometry/io/wkt/wkt.hpp>

#include <boost/foreach.hpp>

int main()
{
    typedef boost::geometry::model::polygon<boost::geometry::model::d2::point_xy<double> > polygon;

    polygon green, blue;

    boost::geometry::read_wkt(
        "POLYGON((2 1.3,2.4 1.7,2.8 1.8,3.4 1.2,3.7 1.6,3.4 2,4.1 3,5.3 2.6,5.4 1.2,4.9 0.8,2.9 ↵
0.7,2 1.3)"
        "(4.0 2.0, 4.2 1.4, 4.8 1.9, 4.4 2.2, 4.0 2.0))", green);

    boost::geometry::read_wkt(
        "POLYGON((4.0 -0.5 , 3.5 1.0 , 2.0 1.5 , 3.5 2.0 , 4.0 3.5 , 4.5 2.0 , 6.0 1.5 , 4.5 ↵
1.0 , 4.0 -0.5))", blue);

    boost::geometry::model::multi_polygon<polygon> multi;

    boost::geometry::sym_difference(green, blue, multi);

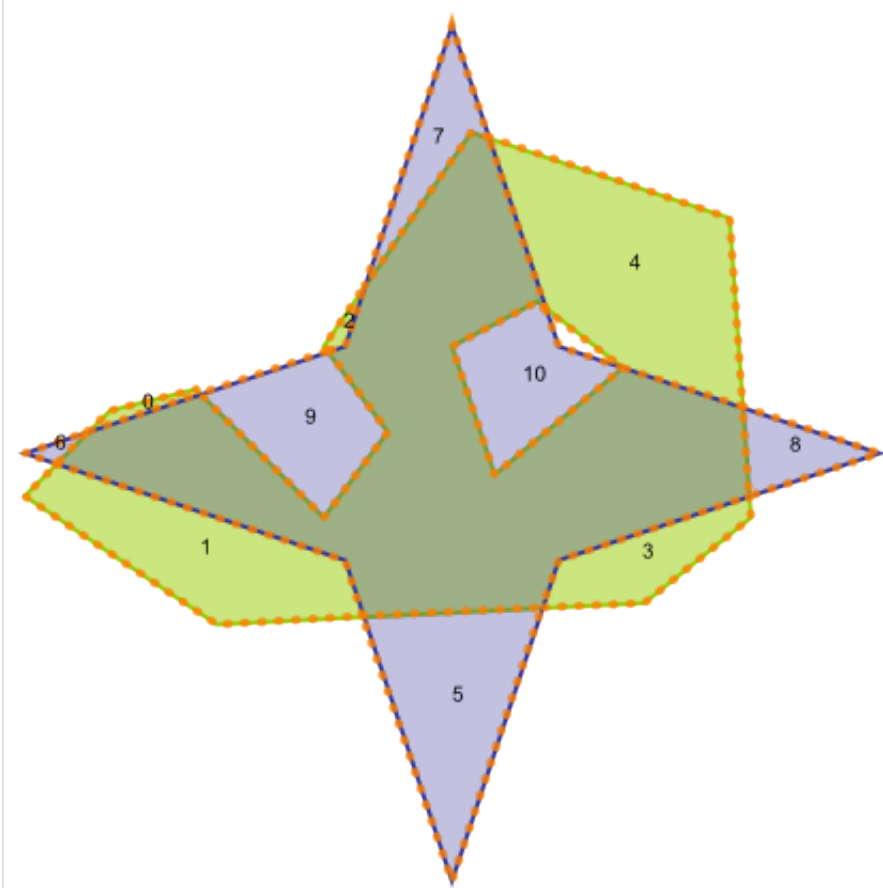
    std::cout
        << "green XOR blue:" << std::endl
        << "total: " << boost::geometry::area(multi) << std::endl;
    int i = 0;
    BOOST_FOREACH(polygon const& p, multi)
    {
        std::cout << i++ << ": " << boost::geometry::area(p) << std::endl;
    }

    return 0;
}
```

Output:

green XOR blue:
total: 3.1459
0: 0.02375
1: 0.542951
2: 0.0149697
3: 0.226855
4: 0.839424
5: 0.525154
6: 0.015
7: 0.181136
8: 0.128798
9: 0.340083
10: 0.307778

The diagram illustrates a complex geometric shape, likely a star or a polygon with internal divisions. The shape is composed of several regions, each labeled with a number from 0 to 10. The regions are colored green, blue, or yellow. The central region is green. The regions are numbered as follows: 0 (yellow, top-left), 1 (yellow, bottom-left), 2 (yellow, top-left), 3 (yellow, bottom-right), 4 (yellow, top-right), 5 (blue, bottom), 6 (blue, top-left), 7 (blue, top), 8 (blue, right), 9 (blue, top-left), and 10 (blue, center). The diagram is surrounded by a dashed orange border.



- difference

- intersection

transform (with strategy)

Transforms from one geometry to another geometry using the specified strategy.

Synopsis

```
template<typename Geometry1, typename Geometry2, typename Strategy>
bool transform(Geometry1 const & geometry1, Geometry2 & geometry2, Strategy const & strategy)
```

Parameters

Type	Concept	Name	Description
Geometry1 const &	Any type fulfilling a Geometry Concept	geometry1	A model of the specified concept
Geometry2 &	Any type fulfilling a Geometry Concept	geometry2	A model of the specified concept
Strategy const &	strategy	strategy	The strategy to be used for transformation

Returns

True if the transformation could be done

Header

Either

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

Or

```
#include <boost/geometry/algorithms/transform.hpp>
```

Complexity

Linear

Example

Shows how points can be scaled, translated or rotated

```

#include <iostream>
#include <boost/geometry.hpp>

int main()
{
    namespace trans = boost::geometry::strategy::transform;
    using boost::geometry::dsv;

    typedef boost::geometry::model::point<double, 2, boost::geometry::cs::cartesian> point_type;

    point_type p1(1.0, 1.0);

    // Translate over (1.5, 1.5)
    point_type p2;
    trans::translate_transformer<point_type, point_type> translate(1.5, 1.5);
    boost::geometry::transform(p1, p2, translate);

    // Scale with factor 3.0
    point_type p3;
    trans::scale_transformer<point_type, point_type> scale(3.0);
    boost::geometry::transform(p1, p3, scale);

    // Rotate with respect to the origin (0,0) over 90 degrees (clockwise)
    point_type p4;
    trans::rotate_transformer<point_type, point_type, boost::geometry::degree> rotate(90.0);
    boost::geometry::transform(p1, p4, rotate);

    std::cout
        << "p1: " << dsv(p1) << std::endl
        << "p2: " << dsv(p2) << std::endl
        << "p3: " << dsv(p3) << std::endl
        << "p4: " << dsv(p4) << std::endl;

    return 0;
}

```

Output:

```

p1: (1, 1)
p2: (2.5, 2.5)
p3: (3, 3)
p4: (1, -1)

```

transform

Transforms from one geometry to another geometry using a strategy.

Synopsis

```

template<typename Geometry1, typename Geometry2>
bool transform(Geometry1 const & geometry1, Geometry2 & geometry2)

```

Parameters

Type	Concept	Name	Description
Geometry1 const &	Any type fulfilling a Geometry Concept	geometry1	A model of the specified concept
Geometry2 &	Any type fulfilling a Geometry Concept	geometry2	A model of the specified concept

Returns

True if the transformation could be done

Header

Either

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

Or

```
#include <boost/geometry/algorithms/transform.hpp>
```

Conformance

The function transform is not defined by OGC.

Behavior

Case	Behavior
Spherical (degree) / Spherical (radian)	Transforms coordinates from degree to radian, or vice versa
Spherical / Cartesian (3D)	Transforms coordinates from spherical coordinates to X,Y,Z, or vice versa, on a unit sphere
Spherical (degree, with radius) / Spherical (radian, with radius)	Transforms coordinates from degree to radian, or vice versa. Third coordinate (radius) is untouched
Spherical (with radius) / Cartesian (3D)	Transforms coordinates from spherical coordinates to X,Y,Z, or vice versa, on a unit sphere. Third coordinate (radius) is taken into account

Complexity

Linear

Example

Shows how points can be transformed using the default strategy


```
#include <iostream>
#include <boost/geometry.hpp>

int main()
{
    namespace bg = boost::geometry;

    // Select a point near the pole (theta=5.0, phi=15.0)
    bg::model::point<long double, 2, bg::cs::spherical<bg::degree> > p1(15.0, 5.0);

    // Transform from degree to radian. Default strategy is automatically selected,
    // it will convert from degree to radian
    bg::model::point<long double, 2, bg::cs::spherical<bg::radian> > p2;
    bg::transform(p1, p2);

    // Transform from degree (lon-lat) to 3D (x,y,z). Default strategy is automatically selected,
    // it will consider points on a unit sphere
    bg::model::point<long double, 3, bg::cs::cartesian> p3;
    bg::transform(p1, p3);

    std::cout
        << "p1: " << bg::dsv(p1) << std::endl
        << "p2: " << bg::dsv(p2) << std::endl
        << "p3: " << bg::dsv(p3) << std::endl;

    return 0;
}
```

Output:

```
p1: (15, 5)
p2: (0.261799, 0.0872665)
p3: (0.084186, 0.0225576, 0.996195)
```

union_

Combines two geometries which each other.

Description

The free function union calculates the spatial set theoretic union of two geometries.

Synopsis

```
template<typename Geometry1, typename Geometry2, typename Collection>
void union_(Geometry1 const & geometry1, Geometry2 const & geometry2, Collection & output_collection)
```

Parameters

Type	Concept	Name	Description
Geometry1 const &	Any type fulfilling a Geometry Concept	geometry1	A model of the specified concept
Geometry2 const &	Any type fulfilling a Geometry Concept	geometry2	A model of the specified concept
Collection &	output collection, either a multi-geometry, or a <code>std::vector<Geometry></code> / <code>std::deque<Geometry></code> etc	output_collection	the output collection

Header

Either

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

Or

```
#include <boost/geometry/algorithms/union.hpp>
```

Conformance

The function `union` implements function `Union` from the [OGC Simple Feature Specification](#).



Note

Boost.Geometry adds an underscore to avoid using the `union` keyword

Behavior

Case	Behavior
GeometryOut is a Polygon	Calculates union polygons of input (multi)polygons and/or boxes



Note

Check the [Polygon Concept](#) for the rules that polygon input for this algorithm should fulfill

Example

Shows how to get a united geometry of two polygons

```

#include <iostream>
#include <vector>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/point_xy.hpp>
#include <boost/geometry/geometries/polygon.hpp>
#include <boost/geometry/io/wkt/wkt.hpp>

#include <boost/foreach.hpp>

int main()
{
    typedef boost::geometry::model::polygon<boost::geometry::model::d2::point_xy<double> > polygon;

    polygon green, blue;

    boost::geometry::read_wkt(
        "POLYGON((2 1.3,2.4 1.7,2.8 1.8,3.4 1.2,3.7 1.6,3.4 2,4.1 3,5.3 2.6,5.4 1.2,4.9 0.8,2.9 ↵
0.7,2 1.3)"
        "(4.0 2.0, 4.2 1.4, 4.8 1.9, 4.4 2.2, 4.0 2.0))", green);

    boost::geometry::read_wkt(
        "POLYGON((4.0 -0.5 , 3.5 1.0 , 2.0 1.5 , 3.5 2.0 , 4.0 3.5 , 4.5 2.0 , 6.0 1.5 , 4.5 ↵
1.0 , 4.0 -0.5))", blue);

    std::vector<polygon> output;
    boost::geometry::union_(green, blue, output);

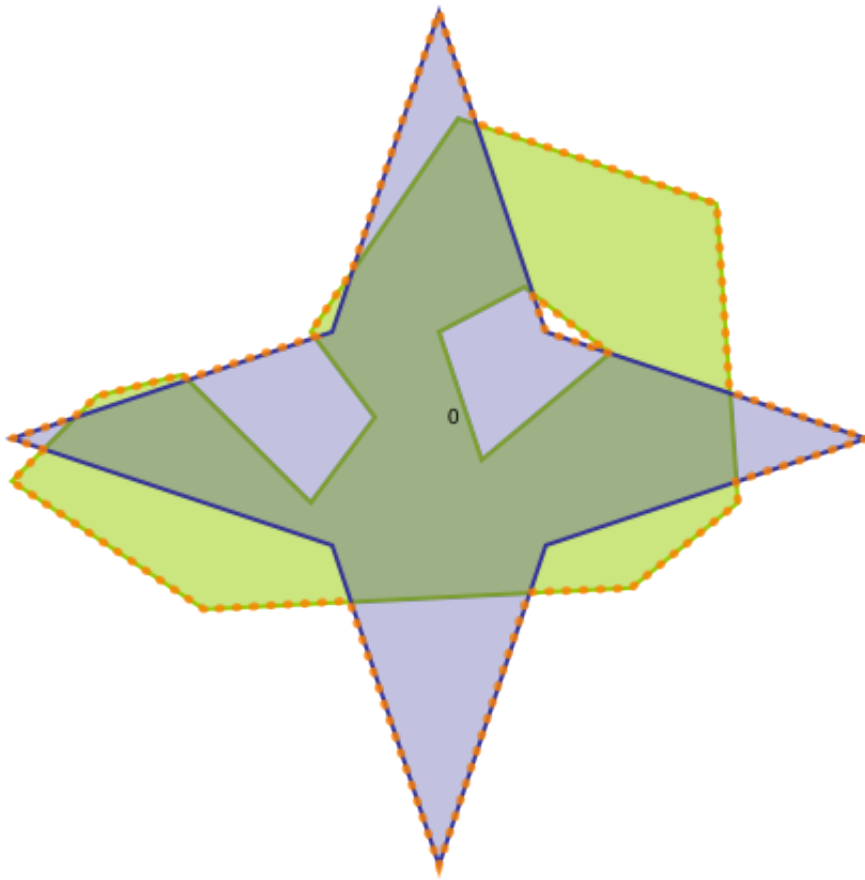
    int i = 0;
    std::cout << "green || blue:" << std::endl;
    BOOST_FOREACH(polygon const& p, output)
    {
        std::cout << i++ << ": " << boost::geometry::area(p) << std::endl;
    }

    return 0;
}

```

Output:

```
green || blue:  
0: 5.64795
```



See also

- [intersection](#)
- [difference](#)
- [sym_difference](#) (symmetric difference)

unique

Calculates the minimal set of a geometry.

Description

The free function `unique` calculates the minimal set (where duplicate consecutive points are removed) of a geometry.

Synopsis

```
template<typename Geometry>  
void unique(Geometry & geometry)
```

Parameters

Type	Concept	Name	Description
Geometry &	Any type fulfilling a Geometry Concept	geometry	A model of the specified concept which will be made unique

Header

Either

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

Or

```
#include <boost/geometry/algorithms/unique.hpp>
```

Conformance

The function `unique` is not defined by OGC.

The function `unique` conforms to the `std::unique` function of the C++ std-library.

Behavior

Case	Behavior
Point	Nothing happens, geometry is unchanged
Segment	Nothing happens, geometry is unchanged
Rectangle	Nothing happens, geometry is unchanged
Linestring	Removes all consecutive duplicate points
Ring	Removes all consecutive duplicate points
Polygon	Removes all consecutive duplicate points in all rings
Multi Point	Nothing happens, geometry is unchanged. Even if two equal points happen to be stored consecutively, they are kept
Multi Linestring	Removes all consecutive duplicate points in all contained lines-trings
Multi Polygon	Removes all consecutive duplicate points in all contained poly-gons (all rings)

Complexity

Linear

Example

Shows how to make a so-called minimal set of a polygon by removing duplicate points

```

#include <iostream>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/polygon.hpp>
#include <boost/geometry/geometries/adapted/boost_tuple.hpp>

BOOST_GEOMETRY_REGISTER_BOOST_TUPLE_CS(cs::cartesian)

int main()
{
    boost::geometry::model::polygon<boost::tuple<double, double> > poly;
    boost::geometry::read_wkt("POLYGON((0 0,0 0 5,5 5,5 5,5 5,5 0,5 0,0 0,0 0,0 0))", poly);
    boost::geometry::unique(poly);
    std::cout << boost::geometry::wkt(poly) << std::endl;

    return 0;
}

```

Output:

```
POLYGON((0 0,0 5,5 5,5 0,0 0))
```

within

within

Checks if the first geometry is completely inside the second geometry.

Description

The free function within checks if the first geometry is completely inside the second geometry.

Synopsis

```

template<typename Geometry1, typename Geometry2>
bool within(Geometry1 const & geometry1, Geometry2 const & geometry2)

```

Parameters

Type	Concept	Name	Description
Geometry1 const &	Any type fulfilling a Geometry Concept	geometry1	A model of the specified concept which might be within the second geometry
Geometry2 const &	Any type fulfilling a Geometry Concept	geometry2	A model of the specified concept which might contain the first geometry

Returns

true if geometry1 is completely contained within geometry2, else false

Header

Either

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

Or

```
#include <boost/geometry/algorithms/within.hpp>
```

Conformance

The function `within` implements function `Within` from the [OGC Simple Feature Specification](#).



Note

OGC defines `within` as completely within and not on the border. See the notes for `within / on the border`

Supported geometries

	Point	Segment	Box	Line-string	Ring	Polygon	Multi-Point	MultiLine-string	MultiPolygon
Point	✗	✗	✗	✗	✗	✗	✗	✗	✗
Segment	✗	✗	✗	✗	✗	✗	✗	✗	✗
Box	✓	✗	✓	✗	✗	✗	✗	✗	✗
Linestring	✗	✗	✗	✗	✗	✗	✗	✗	✗
Ring	✓	✗	✗	✗	✗	✗	✗	✗	✗
Polygon	✓	✗	✗	✗	✗	✗	✗	✗	✗
Multi-Point	✗	✗	✗	✗	✗	✗	✗	✗	✗
MultiLine-string	✗	✗	✗	✗	✗	✗	✗	✗	✗
MultiPolygon	✓	✗	✗	✗	✗	✗	✗	✗	✗



Note

In this status matrix above: columns are types of first parameter and rows are types of second parameter. So a point can be checked to be within a polygon, but not vice versa.

Notes

If a point is located exactly on the border of a geometry, the result depends on the strategy. The default strategy ([Winding \(coordinate system agnostic\)](#)) returns false in that case.

If a polygon has a reverse oriented (e.g. counterclockwise for a clockwise typed polygon), the result also depends on the strategy. The default strategy returns still true if a point is completely within the reversed polygon. There is a specific strategy which returns false in this case.

Complexity

Linear

See also

- [covered_by](#)

Example

Shows how to detect if a point is inside a polygon, or not

```
#include <iostream>
#include <list>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/point_xy.hpp>
#include <boost/geometry/geometries/polygon.hpp>

#include <boost/geometry/io/wkt/wkt.hpp>

int main()
{
    typedef boost::geometry::model::d2::point_xy<double> point_type;
    typedef boost::geometry::model::polygon<point_type> polygon_type;

    polygon_type poly;
    boost::geometry::read_wkt(
        "POLYGON((2 1.3,2.4 1.7,2.8 1.8,3.4 1.2,3.7 1.6,3.4 2,4.1 3,5.3 2.6,5.4 1.2,4.9 0.8,2.9 ↵
0.7,2 1.3)"
        "(4.0 2.0, 4.2 1.4, 4.8 1.9, 4.4 2.2, 4.0 2.0))", poly);

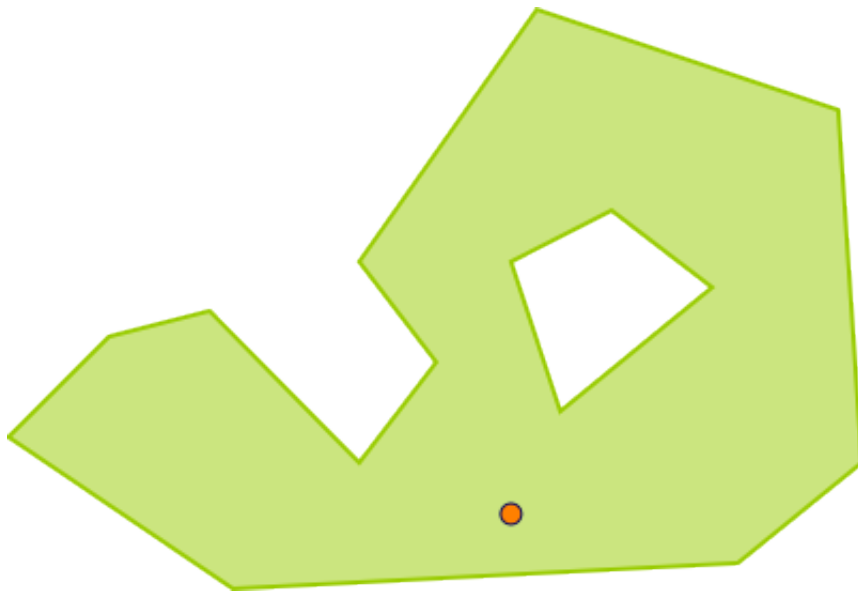
    point_type p(4, 1);

    std::cout << "within: " << (boost::geometry::within(p, poly) ? "yes" : "no") << std::endl;

    return 0;
}
```

Output:

within: yes



within (with strategy)

Checks if the first geometry is completely inside the second geometry using the specified strategy.

Description

The free function `within` checks if the first geometry is completely inside the second geometry, using the specified strategy. Reasons to specify a strategy include: use another coordinate system for calculations; construct the strategy beforehand (e.g. with the radius of the Earth); select a strategy when there are more than one available for a calculation.

Synopsis

```
template<typename Geometry1, typename Geometry2, typename Strategy>
bool within(Geometry1 const & geometry1, Geometry2 const & geometry2, Strategy const & strategy)
```

Parameters

Type	Concept	Name	Description
Geometry1 const &	Any type fulfilling a Geometry Concept	geometry1	A model of the specified concept which might be within the second geometry
Geometry2 const &	Any type fulfilling a Geometry Concept	geometry2	A model of the specified concept which might contain the first geometry
Strategy const &		strategy	strategy to be used

Returns

true if geometry1 is completely contained within geometry2, else false

Header

Either

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

Or

```
#include <boost/geometry/algorithms/within.hpp>
```

Conformance

The function `within` implements function `Within` from the [OGC Simple Feature Specification](#).



Note

OGC defines `within` as completely within and not on the border. See the notes for `within / on the border`

Supported geometries

	Point	Segment	Box	Line-string	Ring	Polygon	Multi-Point	MultiLine-string	MultiPolygon
Point	✗	✗	✗	✗	✗	✗	✗	✗	✗
Segment	✗	✗	✗	✗	✗	✗	✗	✗	✗
Box	✓	✗	✓	✗	✗	✗	✗	✗	✗
Linestring	✗	✗	✗	✗	✗	✗	✗	✗	✗
Ring	✓	✗	✗	✗	✗	✗	✗	✗	✗
Polygon	✓	✗	✗	✗	✗	✗	✗	✗	✗
Multi-Point	✗	✗	✗	✗	✗	✗	✗	✗	✗
MultiLine-string	✗	✗	✗	✗	✗	✗	✗	✗	✗
MultiPolygon	✓	✗	✗	✗	✗	✗	✗	✗	✗



Note

In this status matrix above: columns are types of first parameter and rows are types of second parameter. So a point can be checked to be within a polygon, but not vice versa.

Notes

If a point is located exactly on the border of a geometry, the result depends on the strategy. The default strategy ([Winding \(coordinate system agnostic\)](#)) returns false in that case.

If a polygon has a reverse oriented (e.g. counterclockwise for a clockwise typed polygon), the result also depends on the strategy. The default strategy returns still true if a point is completely within the reversed polygon. There is a specific strategy which returns false in this case.

Complexity

Linear

See also

- [covered_by](#)

Available Strategies

- [Winding](#) (coordinate system agnostic)
- [Franklin](#) (cartesian)
- [Crossings Multiply](#) (cartesian)

Example

[within_strategy] [within_strategy_output]

Arithmetic

add_point

Adds a point to another.

Description

The coordinates of the second point will be added to those of the first point. The second point is not modified.

Synopsis

```
template<typename Point1, typename Point2>
void add_point(Point1 & p1, Point2 const & p2)
```

Parameters

Type	Concept	Name	Description
Point1 &		p1	first point
Point2 const &		p2	second point

Header

Either

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

Or

```
#include <boost/geometry/arithmetic/arithmetic.hpp>
```

add_value

Adds the same value to each coordinate of a point.

Synopsis

```
template<typename Point>
void add_value(Point & p, typename detail::param< Point >::type value)
```

Parameters

Type	Concept	Name	Description
Point &		p	point
typename detail::param< Point >::type		value	value to add

Header

Either

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

Or

```
#include <boost/geometry/arithmetic/arithmetic.hpp>
```

assign_point

Assign a point with another.

Description

The coordinates of the first point will be assigned those of the second point. The second point is not modified.

Synopsis

```
template<typename Point1, typename Point2>
void assign_point(Point1 & p1, const Point2 & p2)
```

Parameters

Type	Concept	Name	Description
Point2		-	Must be specified
Point1 &		p1	first point
const Point2 &		p2	second point

Header

Either

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

Or

```
#include <boost/geometry/arithmetic/arithmetic.hpp>
```

assign_value

Assign each coordinate of a point the same value.

Synopsis

```
template<typename Point>
void assign_value(Point & p, typename detail::param< Point >::type value)
```

Parameters

Type	Concept	Name	Description
Point &		p	point
typename detail::param< Point >::type		value	value to assign

Header

Either

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

Or

```
#include <boost/geometry/arithmetic/arithmetic.hpp>
```

divide_point

Divides a point by another.

Description

The coordinates of the first point will be divided by those of the second point. The second point is not modified.

Synopsis

```
template<typename Point1, typename Point2>
void divide_point(Point1 & p1, Point2 const & p2)
```

Parameters

Type	Concept	Name	Description
Point1 &		p1	first point
Point2 const &		p2	second point

Header

Either

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

Or

```
#include <boost/geometry/arithmetic/arithmetic.hpp>
```

divide_value

Divides each coordinate of the same point by a value.

Synopsis

```
template<typename Point>
void divide_value(Point & p, typename detail::param< Point >::type value)
```

Parameters

Type	Concept	Name	Description
Point &		p	point
typename detail::param< Point >::type		value	value to divide by

Header

Either

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

Or

```
#include <boost/geometry/arithmetic/arithmetic.hpp>
```

dot_product

Computes the dot product (or scalar product) of 2 vectors (points).

Synopsis

```
template<typename P1, typename P2>
select_coordinate_type<P1, P2>::type dot_product(P1 const & p1, P2 const & p2)
```

Parameters

Type	Concept	Name	Description
P1 const &		p1	first point
P2 const &		p2	second point

Returns

the dot product

Header

Either

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

Or

```
#include <boost/geometry/arithmetic/dot_product.hpp>
```

multiply_point

Multiplies a point by another.

Description

The coordinates of the first point will be multiplied by those of the second point. The second point is not modified.

Synopsis

```
template<typename Point1, typename Point2>
void multiply_point(Point1 & p1, Point2 const & p2)
```

Parameters

Type	Concept	Name	Description
Point1 &		p1	first point
Point2 const &		p2	second point

Header

Either

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

Or

```
#include <boost/geometry/arithmetic/arithmetic.hpp>
```

multiply_value

Multiplies each coordinate of a point by the same value.

Synopsis

```
template<typename Point>
void multiply_value(Point & p, typename detail::param< Point >::type value)
```

Parameters

Type	Concept	Name	Description
Point &		p	point
typename detail::param< Point >::type		value	value to multiply by

Header

Either

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

Or

```
#include <boost/geometry/arithmetic/arithmetic.hpp>
```

subtract_point

Subtracts a point to another.

Description

The coordinates of the second point will be subtracted to those of the first point. The second point is not modified.

Synopsis

```
template<typename Point1, typename Point2>
void subtract_point(Point1 & p1, Point2 const & p2)
```

Parameters

Type	Concept	Name	Description
Point1 &		p1	first point
Point2 const &		p2	second point

Header

Either

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

Or

```
#include <boost/geometry/arithmetic/arithmetic.hpp>
```

subtract_value

Subtracts the same value to each coordinate of a point.

Synopsis

```
template<typename Point>
void subtract_value(Point & p, typename detail::param< Point >::type value)
```

Parameters

Type	Concept	Name	Description
Point &		p	point
typename detail::param< Point >::type		value	value to subtract

Header

Either

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

Or

```
#include <boost/geometry/arithmetic/arithmetic.hpp>
```

Concepts

Point Concept

Description

The Point Concept describes the requirements for a point type. All algorithms in Boost.Geometry will check any geometry arguments against the concept requirements.

A point is *an entity that has a location in space or on a plane, but has no extent* ([wiki](#)). The point is the most basic geometry of Boost.Geometry, most other geometries consist of points. (*Exceptions are box and segment, which might consist of two points but that is not necessarily the case.*)

Concept Definition

The Point Concept is defined as following:

- there must be a specialization of `traits::tag`, defining `point_tag` as type
- there must be a specialization of `traits::coordinate_type`, defining the type of its coordinates
- there must be a specialization of `traits::coordinate_system`, defining its coordinate system (cartesian, spherical, etc)
- there must be a specialization of `traits::dimension`, defining its number of dimensions (2, 3, ...) (hint: derive it conveniently from `boost::mpl::int_<X>` for X Dimensional)
- there must be a specialization of `traits::access`, per dimension, with two functions:
 - `get` to get a coordinate value
 - `set` to set a coordinate value (this one is not checked for `ConstPoint`)

Available Models

- `model::point`

- [model::d2::point_xy](#)
- a lat long point (currently in an extension)
- [C array](#)
- [Boost.Array](#)
- [Boost.Fusion](#)
- [Boost.Polygon](#)
- [Boost.Tuple](#)
- other point types, adapted e.g. using one of the [registration macro's](#)

Linestring Concept

Description

The Linestring Concept describes the requirements for a linestring type. All algorithms in Boost.Geometry will check any geometry arguments against the concept requirements.

A linestring is *a Curve with linear interpolation between Points*. ([OGC Simple Feature Specification](#)).

Concept Definition

The Linestring Concept is defined as following:

- there must be a specialization of `traits::tag` defining `linestring_tag` as type
- it must behave like a `Boost.Range Random Access Range`
- The type defined by the metafunction `range_value<...>::type` must fulfill the [Point Concept](#)

Rules

Besides the Concepts, which are checks on compile-time, there are rules that valid linestrings must fulfill. Most algorithms work on any linestring, so either self-crossing or not. However, for correct results using the overlay algorithms (intersection and difference algorithms in combination with a polygon) self-intersections can disturb the process and result in incorrect results.

Available Models

- [model::linestring](#)
- a `std::vector` (requires registration)
- a `std::deque` (requires registration)



Note

See also the sample in the [Boost.Range documentation](#) showing how a type can be adapted to a `Boost.Range` to fulfill the concept of a Linestring

Polygon Concept

Description

The Polygon Concept describes the requirements for a polygon type. All algorithms in Boost.Geometry will check any geometry arguments against the concept requirements.

A polygon is *A polygon is a planar surface defined by one exterior boundary and zero or more interior boundaries* ([OGC Simple Feature Specification](#)).

So the definition of a Boost.Geometry polygon differs a bit from e.g. Wiki, where a polygon does not have holes. A polygon of Boost.Geometry is a polygon with or without holes. (*A polygon without holes is a helper geometry within Boost.Geometry, and referred to as a ring.*)

Concept Definition

The Polygon Concept is defined as following:

- there must be a specialization of `traits::tag` defining `polygon_tag` as type
- there must be a specialization of `traits::ring_type` defining the type of its exterior ring and interior rings as type
- this type defined by `ring_type` must fulfill the [Ring Concept](#)
- there must be a specialization of `traits::interior_type` defining the type of the collection of its interior rings as type; this collection itself must fulfill a Boost.Range Random Access Range Concept
- there must be a specialization of `traits::exterior_ring` with two functions named `get`, returning the exterior ring, one being `const`, the other being `non const`
- there must be a specialization of `traits::interior_rings` with two functions named `get`, returning the interior rings, one being `const`, the other being `non const`

Rules

Besides the Concepts, which are checks on compile-time, there are some other rules that valid polygons must fulfill. This follows the [opengeospatial](#) rules (see link above).

- Polygons are simple geometric objects (See also [wiki](#) but holes are allowed in Boost.Geometry polygons).
- If the polygons underlying `ring_type` is defined as clockwise, the exterior ring must have the clockwise orientation, and any interior ring must be counter clockwise. If the `ring_type` is defined counter clockwise, it is vice versa.
- If the polygons underlying `ring_type` is defined as closed, all rings must be closed: the first point must be spatially equal to the last point.
- The interior is a connected point set.
- There should be no self intersections, but self tangencies (between exterior/interior rings) are allowed (as long as the interior is a connected point set).
- There should be no cut lines, spikes or punctures.
- The interior rings should be located within the exterior ring. Interior rings may not be located within each other.

The algorithms such as intersection, area, centroid, union, etc. do not check validity. There will be an algorithm `is_valid` which checks for validity against these rules, at runtime, and which can be called (by the library user) before.

If the input is invalid, the output might be invalid too. For example: if a polygon which should be closed is not closed, the area will be incorrect.

Available Models

- [polygon](#)
- a Boost.Polygon `polygon_with_holes_data` (requires `#include boost/geometry/geometries/adapted/boost_polygon/polygon.hpp`)

MultiPoint Concept

Description

The MultiPoint Concept describes the requirements for a multi point type. All algorithms in Boost.Geometry will check any geometry arguments against the concept requirements.

Concept Definition

The MultiPoint Concept is defined as following:

- There must be a specialization of the metafunction `traits::tag`, defining `multi_point_tag` as type
- It must behave like a Boost.Range Random Access Range
- The type defined by the metafunction `range_value<...>::type` must fulfill the [Point Concept](#)

Available Models

- `model::multi_point`

MultiLinestring Concept

Description

The MultiLinestring Concept describes the requirements for a multi linestring type. All algorithms in Boost.Geometry will check any geometry arguments against the concept requirements.

Concept Definition

The MultiLinestring Concept is defined as following:

- There must be a specialization of the metafunction `traits::tag`, defining `multi_linestring_tag` as type
- It must behave like a Boost.Range Random Access Range
- The type defined by the metafunction `range_value<...>::type` must fulfill the [Linestring Concept](#)

Available Models

- `model::multi_linestring`

MultiPolygon Concept

Description

The MultiPolygon Concept describes the requirements for a multi polygon type. All algorithms in Boost.Geometry will check any geometry arguments against the concept requirements.

Concept Definition

The MultiPolygon Concept is defined as following:

- There must be a specialization of the metafunction `traits::tag`, defining `multi_polygon_tag` as type
- It must behave like a Boost.Range Random Access Range
- The type defined by the metafunction `range_value<...>::type` must fulfill the [Polygon Concept](#)

Rules

Besides the Concepts, which are checks on compile-time, there are rules that valid MultiPolygons must fulfill. See the [Polygon Concept](#) for more information on the rules a polygon (and also a multi polygon) must fulfill.

Additionally:

- Individual polygons making up a multi-polygon may not intersect each other, but tangencies are allowed.
- One polygon might be located within the interior ring of another polygon.

Available Models

- `model::multi_polygon`

Box Concept

Description

The Box Concept describes the requirements for a box type. All algorithms in Boost.Geometry will check any geometry arguments against the concept requirements.

A box is a geometry with (usually) two or three dimensions, having its axis aligned to the coordinate system.

The box is not one of the basic types in Boost.Geometry (point, linestring, polygon) but it is a *helper type*. The main reasons for the box existence are its usefulness for indexing (a spatial index, or splitting a geometry into monotonic sections) and it is the output of the [envelope](#) algorithm.

Therefore, a box is axis aligned (the envelope is also called aabb, axis aligned bounding box).

Concept Definition

The Box Concept is defined as following:

- there must be a specialization of `traits::tag`, defining `box_tag` as type
- there must be a specialization of `traits::point_type` to define the underlying point type (even if it does not consist of points, it should define this type, to indicate the points it can work with)
- there must be a specialization of `traits::indexed_access`, per index (`min_corner`, `max_corner`) and per dimension, with two functions:
 - `get` to get a coordinate value
 - `set` to set a coordinate value (this one is not checked for `ConstBox`)

Available Models

- `model::box`

Ring Concept

Description

The Ring Concept describes the requirements for a ring type. All algorithms in Boost.Geometry will check any geometry arguments against the concept requirements.



Note

Also called linear ring, but we explicitly refer to a filled feature here

Concept Definition

The Ring Concept is defined as following:

- there must be a specialization of `traits::tag` defining `ring_tag` as type
- it must behave like a `Boost.Range Random Access Range`
- The type defined by the metafunction `range_value<...>::type` must fulfill the [Point Concept](#)
- there might be a specialization of `traits::point_order` defining the order or orientation of its points, `clockwise` or `counterclockwise`
- there might be a specialization of `traits::closure` defining the closure, `open` or `closed`

Rules

Besides the Concepts, which are checks on compile-time, there are rules that valid rings must fulfill. See the [Polygon Concept](#) for more information on the rules a polygon (and also a ring) must fulfill.

Available Models

- [ring](#)
- a `Boost.Polygon polygon_data` (requires `#include boost/geometry/geometries/adapted/boost_polygon/ring.hpp`)
- a `std::vector` (requires `#include boost/geometry/geometries/adapted/std_as_ring.hpp`)
- a `std::deque` (requires `#include boost/geometry/geometries/adapted/std_as_ring.hpp`)



Note

See also the sample in the [Boost.Range documentation](#) showing how a type can be adapted to a `Boost.Range` to fulfill the concept of a Ring

Segment Concept

Description

The Segment Concept describes the requirements for a segment type. All algorithms in `Boost.Geometry` will check any geometry arguments against the concept requirements.

Concept Definition

- there must be a specialization of `traits::tag` defining `segment_tag` as type
- there must be a specialization of `traits::point_type` to define the underlying point type (even if it does not consist of points, it should define this type, to indicate the points it can work with)
- there must be a specialization of `traits::indexed_access`, per index and per dimension, with two functions:
 - `get` to get a coordinate value
 - `set` to set a coordinate value (this one is not checked for `ConstSegment`)

**Note**

The segment concept is similar to the box concept, defining using another tag. However, the box concept assumes the index as `min_corner`, `max_corner`, while for the segment concept, there is no assumption.

Available Models

- `model::segment`
- `referring segment`

Constants

`min_corner`

Indicates the minimal corner (lower left) of a box to be get, set or processed

Synopsis

```
int const min_corner = 0;
```

Header

Either

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

Or

```
#include <boost/geometry/core/access.hpp>
```

**Note**

`min_corner` and `max_corner` are only applicable for boxes and not for, e.g., a segment

**Note**

`min_corner` should be the minimal corner of a box, but that is not guaranteed. Use `correct` to make `min_corner` the minimal corner. The same applies for `max_corner`.

Example

Get the coordinate of a box

```

#include <iostream>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/point_xy.hpp>

namespace bg = boost::geometry;

int main()
{
    bg::model::box<bg::model::d2::point_xy<double> > box;

    bg::assign_values(box, 1, 3, 5, 6);

    std::cout << "Box: "
        << " " << bg::get<bg::min_corner, 0>(box)
        << " " << bg::get<bg::min_corner, 1>(box)
        << " " << bg::get<bg::max_corner, 0>(box)
        << " " << bg::get<bg::max_corner, 1>(box)
        << std::endl;

    return 0;
}

```

Output:

```
Box: 1 3 5 6
```

See also

- [max_corner](#)
- [get with index](#)
- [set with index](#)

max_corner

Indicates the maximal corner (upper right) of a box to be get, set or processed

Synopsis

```
int const max_corner = 1;
```

Header

Either

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

Or

```
#include <boost/geometry/core/access.hpp>
```



Note

min_corner and max_corner are only applicable for boxes and not for, e.g., a segment



Note

`min_corner` should be the minimal corner of a box, but that is not guaranteed. Use `correct` to make `min_corner` the minimal corner. The same applies for `max_corner`.

Example

Get the coordinate of a box

```
#include <iostream>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/point_xy.hpp>

namespace bg = boost::geometry;

int main()
{
    bg::model::box<bg::model::d2::point_xy<double> > box;

    bg::assign_values(box, 1, 3, 5, 6);

    std::cout << "Box:"
        << " " << bg::get<bg::min_corner, 0>(box)
        << " " << bg::get<bg::min_corner, 1>(box)
        << " " << bg::get<bg::max_corner, 0>(box)
        << " " << bg::get<bg::max_corner, 1>(box)
        << std::endl;

    return 0;
}
```

Output:

```
Box: 1 3 5 6
```

See also

- [min_corner](#)
- [get with index](#)
- [set with index](#)

Coordinate Systems

cs::cartesian

Cartesian coordinate system.

Description

Defines the Cartesian or rectangular coordinate system where points are defined in 2 or 3 (or more) dimensions and usually (but not always) known as x,y,z

Synopsis

```
struct cs::cartesian
{
    // ...
};
```

Header

Either

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

Or

```
#include <boost/geometry/core/cs.hpp>
```

cs::spherical

Spherical (polar) coordinate system, in degree or in radian.

Description

Defines the spherical coordinate system where points are defined in two angles and an optional radius usually known as r, theta, phi

Synopsis

```
template<typename DegreeOrRadian>
struct cs::spherical
{
    // ...
};
```

Template parameter(s)

Parameter	Description
typename DegreeOrRadian	

Header

Either

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

Or

```
#include <boost/geometry/core/cs.hpp>
```

cs::spherical_equatorial

Spherical equatorial coordinate system, in degree or in radian.

Description

This one resembles the geographic coordinate system, and has latitude up from zero at the equator, to 90 at the pole (opposite to the spherical(polar) coordinate system). Used in astronomy and in GIS (but there is also the geographic)

Synopsis

```
template<typename DegreeOrRadian>
struct cs::spherical_equatorial
{
    // ...
};
```

Template parameter(s)

Parameter	Description
typename DegreeOrRadian	

Header

Either

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

Or

```
#include <boost/geometry/core/cs.hpp>
```

cs::geographic

Geographic coordinate system, in degree or in radian.

Description

Defines the geographic coordinate system where points are defined in two angles and usually known as lat,long or lo,la or phi,lambda

Synopsis

```
template<typename DegreeOrRadian>
struct cs::geographic
{
    // ...
};
```

Template parameter(s)

Parameter	Description
typename DegreeOrRadian	

Header

Either

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

Or

```
#include <boost/geometry/core/cs.hpp>
```

Core Metafunctions

closure

Metafunction defining **value** as the closure (clockwise, counterclockwise) of the specified geometry type.

Synopsis

```
template<typename Geometry>
struct closure
{
    // ...
};
```

Template parameter(s)

Parameter	Description
typename Geometry	Any type fulfilling a Geometry Concept

Header

```
#include <boost/geometry/core/closure.hpp>
```



Note

The closure is defined for any geometry type, but only has a real meaning for areal geometry types (ring, polygon, multi_polygon)

Complexity

Compile time

Example

Examine if a polygon is defined as "should be closed"

```
#include <iostream>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/polygon.hpp>
#include <boost/geometry/geometries/point_xy.hpp>

int main()
{
    typedef boost::geometry::model::d2::point_xy<double> point_type;
    typedef boost::geometry::model::polygon<point_type> polygon_type;

    boost::geometry::closure_selector clos = boost::geometry::closure<polygon_type>::value;

    std::cout << "closure: " << clos << std::endl
        << "(open = " << boost::geometry::open
        << ", closed = " << boost::geometry::closed
        << ")" << std::endl;

    return 0;
}
```

Output:

```
closure: 1
(open = 0, closed = 1)
```

See also

- [The closure_selector enumeration](#)

coordinate_system

Metafunction defining **type** as the coordinate system (cartesian, spherical, etc) of the point type making up the specified geometry type.

Synopsis

```
template<typename Geometry>
struct coordinate_system
{
    // ...
};
```

Template parameter(s)

Parameter	Description
typename Geometry	Any type fulfilling a Geometry Concept

Header

```
#include <boost/geometry/core/coordinate_system.hpp>
```

Complexity

Compile time

Example

Examine the coordinate system of a point

```
#include <iostream>
#include <typeinfo>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/polygon.hpp>
#include <boost/geometry/geometries/point_xy.hpp>

int main()
{
    typedef boost::geometry::model::d2::point_xy<double> point_type;
    typedef boost::geometry::model::polygon<point_type> polygon_type;

    typedef boost::geometry::coordinate_system<polygon_type>::type system;

    std::cout << "system: " << typeid(system).name() << std::endl;

    return 0;
}
```

Output (using MSVC):

```
system: struct boost::geometry::cs::cartesian
```

coordinate_type

Metafunction defining **type** as the coordinate type (int, float, double, etc) of the point type making up the specified geometry type.

Synopsis

```
template<typename Geometry>
struct coordinate_type
{
    // ...
};
```

Template parameter(s)

Parameter	Description
typename Geometry	Any type fulfilling a Geometry Concept

Header

```
#include <boost/geometry/core/coordinate_type.hpp>
```

Complexity

Compile time

Example

Examine the coordinate type of a point

```
#include <iostream>
#include <typeinfo>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/polygon.hpp>
#include <boost/geometry/geometries/point_xy.hpp>

int main()
{
    typedef boost::geometry::model::d2::point_xy<double> point_type;
    typedef boost::geometry::model::polygon<point_type> polygon_type;

    typedef boost::geometry::coordinate_type<polygon_type>::type ctype;

    std::cout << "type: " << typeid(ctype).name() << std::endl;

    return 0;
}
```

Output (using MSVC):

```
type: double
```

cs_tag

Meta-function returning coordinate system tag (cs family) of any geometry.

Synopsis

```
template<typename G>
struct cs_tag
{
    // ...
};
```

Template parameter(s)

Parameter	Description
typename G	

Header

Either

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

Or

```
#include <boost/geometry/core/cs.hpp>
```

degree

Unit of plane angle: Degrees.

Description

Tag defining the unit of plane angle for spherical coordinate systems. This tag specifies that coordinates are defined in degrees (-180 .. 180). It has to be specified for some coordinate systems.

Synopsis

```
struct degree
{
    // ...
};
```

Header

Either

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

Or

```
#include <boost/geometry/core/cs.hpp>
```

example

Specify two coordinate systems, one in degrees, one in radians.

```

#include <iostream>
#include <boost/geometry.hpp>

using namespace boost::geometry;

int main()
{
    typedef model::point<double, 2, cs::spherical_equatorial<degree> > degree_point;
    typedef model::point<double, 2, cs::spherical_equatorial<radian> > radian_point;

    degree_point d(4.893, 52.373);
    radian_point r(0.041, 0.8527);

    double dist = distance(d, r);
    std::cout
        << "distance:" << std::endl
        << dist << " over unit sphere" << std::endl
        << dist * 3959 << " over a spherical earth, in miles" << std::endl;

    return 0;
}

```

Output:

```

distance:
0.0675272 over unit sphere
267.34 over a spherical earth, in miles

```

dimension

Metafunction defining **value** as the number of coordinates (the number of axes of any geometry) of the point type making up the specified geometry type.

Synopsis

```

template<typename Geometry>
struct dimension
{
    // ...
};

```

Template parameter(s)

Parameter	Description
typename Geometry	Any type fulfilling a Geometry Concept

Header

```
#include <boost/geometry/core/coordinate_dimension.hpp>
```

Complexity

Compile time

Example

Examine the number of coordinates making up the points in a linestring type


```

#include <iostream>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/linestring.hpp>
#include <boost/geometry/geometries/adapted/boost_tuple.hpp>

BOOST_GEOMETRY_REGISTER_BOOST_TUPLE_CS(cs::cartesian);

int main()
{
    int dim = boost::geometry::dimension
        <
            boost::geometry::model::linestring
                <
                    boost::tuple<float, float, float>
                >
            >::value;

    std::cout << "dimensions: " << dim << std::endl;

    return 0;
}

```

Output:

```
dimensions: 3
```

interior_type

Metafunction defining **type** as the `interior_type` (container type of inner rings) of the specified geometry type.

Description

Interior rings should be organized as a container (`std::vector`, `std::deque`, `boost::array`) with `Boost.Range` support. This metafunction defines the type of the container.

Synopsis

```

template<typename Geometry>
struct interior_type
{
    // ...
};

```

Template parameter(s)

Parameter	Description
typename Geometry	A type fulfilling the Polygon or MultiPolygon concept.

Header

```
#include <boost/geometry/core/interior_type.hpp>
```

Complexity

Compile time

Example

Shows how to use the `interior_type` metafunction

```
#include <iostream>
#include <typeinfo>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/polygon.hpp>
#include <boost/geometry/geometries/ring.hpp>
#include <boost/geometry/geometries/adapted/boost_array.hpp>

BOOST_GEOMETRY_REGISTER_BOOST_ARRAY_CS(cs::cartesian)

int main()
{
    // Define a polygon storing points in a deque and storing interior rings
    // in a list (note that std::list is not supported by most algorithms
    // because not supporting a random access iterator)
    typedef boost::geometry::model::polygon
        <
            boost::array<short, 3>,
            true, true,
            std::deque, std::list
        > polygon;

    std::cout << typeid(boost::geometry::interior_type<polygon>::type).name() << std::endl;

    return 0;
}
```

Output (using MSVC) is a long story (part manually replaced with ellipsis):

```
class std::list<class boost::geometry::model::ring<class boost::array<short,3>,1,1,class ↵
std::deque,class std::allocator>,class std::allocator<...> > >
```

is_radian

Meta-function to verify if a coordinate system is radian.

Synopsis

```
template<typename CoordinateSystem>
struct is_radian
{
    // ...
};
```

Template parameter(s)

Parameter	Description
typename CoordinateSystem	

Header

Either

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

Or

```
#include <boost/geometry/core/cs.hpp>
```

point_order

Metafunction defining **value** as the point order (clockwise, counterclockwise) of the specified geometry type.

Synopsis

```
template<typename Geometry>
struct point_order
{
    // ...
};
```

Template parameter(s)

Parameter	Description
typename Geometry	Any type fulfilling a Geometry Concept

Header

```
#include <boost/geometry/core/point_order.hpp>
```



Note

The point order is defined for any geometry type, but only has a real meaning for areal geometry types (ring, polygon, multi_polygon)

Complexity

Compile time

Example

Examine the expected point order of a polygon type

```

#include <iostream>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/polygon.hpp>
#include <boost/geometry/geometries/point_xy.hpp>

int main()
{
    typedef boost::geometry::model::d2::point_xy<double> point_type;
    typedef boost::geometry::model::polygon<point_type, false> polygon_type;

    boost::geometry::order_selector order = boost::geometry::point_order<polygon_type>::value;

    std::cout << "order: " << order << std::endl
              << "(clockwise = " << boost::geometry::clockwise
              << ", counterclockwise = " << boost::geometry::counterclockwise
              << ") " << std::endl;

    return 0;
}

```

Output:

```

order: 2
(clockwise = 1, counterclockwise = 2)

```

See also

- [The order_selector enumeration](#)

point_type

Metafunction defining **type** as the point_type of the specified geometry type.

Synopsis

```

template<typename Geometry>
struct point_type
{
    // ...
};

```

Template parameter(s)

Parameter	Description
typename Geometry	Any type fulfilling a Geometry Concept

Header

```
#include <boost/geometry/core/point_type.hpp>
```

Complexity

Compile time

Example

Examine the point type of a multi_polygon

```
#include <iostream>
#include <typeinfo>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/polygon.hpp>
#include <boost/geometry/geometries/point_xy.hpp>
#include <boost/geometry/multi/geometries/multi_polygon.hpp>

int main()
{
    typedef boost::geometry::model::d2::point_xy<double> point_type;
    typedef boost::geometry::model::polygon<point_type> polygon_type;
    typedef boost::geometry::model::multi_polygon<polygon_type> mp_type;

    typedef boost::geometry::point_type<mp_type>::type ptype;

    std::cout << "point type: " << typeid(ptype).name() << std::endl;

    return 0;
}
```

Output (in MSVC):

```
point type: class boost::geometry::model::d2::point_xy<double,struct boost::geo-
metry::cs::cartesian>
```

radian

Unit of plane angle: Radians.

Description

Tag defining the unit of plane angle for spherical coordinate systems. This tag specifies that coordinates are defined in radians (-PI .. PI). It has to be specified for some coordinate systems.

Synopsis

```
struct radian
{
    // ...
};
```

Header

Either

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

Or

```
#include <boost/geometry/core/cs.hpp>
```

example

Specify two coordinate systems, one in degrees, one in radians.

```

#include <iostream>
#include <boost/geometry.hpp>

using namespace boost::geometry;

int main()
{
    typedef model::point<double, 2, cs::spherical_equatorial<degree> > degree_point;
    typedef model::point<double, 2, cs::spherical_equatorial<radian> > radian_point;

    degree_point d(4.893, 52.373);
    radian_point r(0.041, 0.8527);

    double dist = distance(d, r);
    std::cout
        << "distance:" << std::endl
        << dist << " over unit sphere" << std::endl
        << dist * 3959 << " over a spherical earth, in miles" << std::endl;

    return 0;
}

```

Output:

```

distance:
0.0675272 over unit sphere
267.34 over a spherical earth, in miles

```

ring_type

Metafunction defining **type** as the `ring_type` of the specified geometry type.

Description

A polygon contains one exterior ring and zero or more interior rings (holes). This metafunction retrieves the type of the rings. Exterior ring and each of the interior rings all have the same `ring_type`.

Synopsis

```

template<typename Geometry>
struct ring_type
{
    // ...
};

```

Template parameter(s)

Parameter	Description
typename Geometry	A type fulfilling the Ring, Polygon or MultiPolygon concept.

Header

```
#include <boost/geometry/core/ring_type.hpp>
```

Complexity

Compile time

Example

Shows how to use the `ring_type` metafunction, as well as `interior_type`

```
#include <iostream>
#include <typeinfo>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/polygon.hpp>
#include <boost/geometry/geometries/point_xy.hpp>

int main()
{
    typedef boost::geometry::model::d2::point_xy<double> point;
    typedef boost::geometry::model::polygon<point> polygon;

    typedef boost::geometry::ring_type<polygon>::type ring_type;
    typedef boost::geometry::interior_type<polygon>::type int_type;

    std::cout << typeid(ring_type).name() << std::endl;
    std::cout << typeid(int_type).name() << std::endl;

    // So int_type defines a collection of rings,
    // which is a Boost.Range compatible range
    // The type of an element of the collection is the very same ring type again.
    // We show that.
    typedef boost::range_value<int_type>::type int_ring_type;

    std::cout
        << std::boolalpha
        << boost::is_same<ring_type, int_ring_type>::value
        << std::endl;

    return 0;
}
```

Output (using gcc):

```
N5boost8geometry5model4ringINS1_2d28point_xyIdNS0_2cs9cartesianEEELb1ELb1ES6vectorSaEE
St6vectorIN5boost8geometry5model4ringINS2_2d28point_xyIdNS1_2cs9cartesianEEELb1ELb1ES_SaEESaIS9_EE
true
```

tag

Metafunction defining **type** as the tag of the specified geometry type.

Description

With Boost.Geometry, tags are the driving force of the tag dispatching mechanism. The tag metafunction is therefore used in every free function.

Synopsis

```
template<typename Geometry>
struct tag
{
    // ...
};
```

Template parameter(s)

Parameter	Description
typename Geometry	Any type fulfilling a Geometry Concept

Header

Either

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

Or

```
#include <boost/geometry/core/tag.hpp>
```

Metafunction result type

The metafunction tag defines **type** as one of the following tags:

- point_tag
- linestring_tag
- polygon_tag
- multi_point_tag
- multi_linestring_tag
- multi_polygon_tag
- box_tag
- segment_tag
- ring_tag

Complexity

Compile time

Example

Shows how tag dispatching essentially works in Boost.Geometry


```

#include <iostream>

#include <boost/assign.hpp>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/polygon.hpp>
#include <boost/geometry/multi/geometries/multi_polygon.hpp>
#include <boost/geometry/geometries/adapted/boost_tuple.hpp>

BOOST_GEOMETRY_REGISTER_BOOST_TUPLE_CS(cs::cartesian)

template <typename Tag> struct dispatch {};

// Specialization for points
template <> struct dispatch<boost::geometry::point_tag>
{
    template <typename Point>
    static inline void apply(Point const& p)
    {
        // Use the Boost.Geometry free function "get"
        // working on all supported point types
        std::cout << "Hello POINT, you are located at: "
            << boost::geometry::get<0>(p) << ", "
            << boost::geometry::get<1>(p)
            << std::endl;
    }
};

// Specialization for polygons
template <> struct dispatch<boost::geometry::polygon_tag>
{
    template <typename Polygon>
    static inline void apply(Polygon const& p)
    {
        // Use the Boost.Geometry manipulator "dsv"
        // working on all supported geometries
        std::cout << "Hello POLYGON, you look like: "
            << boost::geometry::dsv(p)
            << std::endl;
    }
};

// Specialization for multipolygons
template <> struct dispatch<boost::geometry::multi_polygon_tag>
{
    template <typename MultiPolygon>
    static inline void apply(MultiPolygon const& m)
    {
        // Use the Boost.Range free function "size" because all
        // multigeometries comply to Boost.Range
        std::cout << "Hello MULTIPOLYGON, you contain: "
            << boost::size(m) << " polygon(s)"
            << std::endl;
    }
};

template <typename Geometry>
inline void hello(Geometry const& geometry)
{
    // Call the metafunction "tag" to dispatch, and call method (here "apply")
    dispatch
    <
        typename boost::geometry::tag<Geometry>::type

```

```

        >::apply(geometry);
    }

int main()
{
    // Define polygon type (here: based on a Boost.Tuple)
    typedef boost::geometry::model::polygon<boost::tuple<int, int> > polygon_type;

    // Declare and fill a polygon and a multipolygon
    polygon_type poly;
    boost::geometry::exterior_ring(poly) = boost::assign::tuple_list_of(0, 0)(0, 10)(10, 5)(0, 0);

    boost::geometry::model::multi_polygon<polygon_type> multi;
    multi.push_back(poly);

    // Call "hello" for point, polygon, multipolygon
    hello(boost::make_tuple(2, 3));
    hello(poly);
    hello(multi);

    return 0;
}

```

Output:

```

Hello POINT, you are located at: 2, 3
Hello POLYGON, you look like: (((0, 0), (0, 10), (10, 5), (0, 0)))
Hello MULTIPOLYGON, you contain: 1 polygon(s)

```

tag_cast

Metafunction defining a type being either the specified tag, or one of the specified basetags if the type inherits from them.

Description

Tags can inherit each other. A multi_point inherits, for example, both the multi_tag and the pointlike tag. Often behaviour can be shared between different geometry types. A tag, found by the metafunction tag, can be casted to a more basic tag, and then dispatched by that tag.

Synopsis

```

template<typename Tag, typename BaseTag, typename BaseTag2, typename BaseTag3, typename BaseTag4,
typename BaseTag5, typename BaseTag6, typename BaseTag7>
struct tag_cast
{
    // ...
};

```

Template parameter(s)

Parameter	Default	Description
typename Tag		The tag to be casted to one of the base tags
typename BaseTag		First base tag
typename BaseTag2	void	Optional second base tag
typename BaseTag3	void	Optional third base tag
typename BaseTag4	void	Optional fourth base tag
typename BaseTag5	void	Optional fifth base tag
typename BaseTag6	void	Optional sixth base tag
typename BaseTag7	void	Optional seventh base tag

Header

Either

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

Or

```
#include <boost/geometry/core/tag_cast.hpp>
```

**Note**

The specified tag list is handled in the specified order: as soon as a tag inheriting the specified tag is found, it is defined as the metafunction typedef **type**.

**Note**

If none of the specified possible base tags is a base class of the specified tag, the tag itself is defined as the **type** result of the metafunction.

Complexity

Compile time

Example

Check if the `polygon_tag` can be casted to the `areal_tag`

```

#include <iostream>
#include <typeinfo>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/polygon.hpp>
#include <boost/geometry/geometries/point_xy.hpp>

namespace geo = boost::geometry;
int main()
{
    typedef geo::model::d2::point_xy<double> point_type;
    typedef geo::model::polygon<point_type> polygon_type;

    typedef geo::tag<polygon_type>::type tag;
    typedef geo::tag_cast<tag, geo::linear_tag, geo::areal_tag>::type base_tag;

    std::cout << "tag: " << typeid(tag).name() << std::endl
               << "base tag: " << typeid(base_tag).name() << std::endl;

    return 0;
}

```

Output (in MSVC):

```

tag: struct boost::geometry::polygon_tag
base tag: struct boost::geometry::areal_tag

```

Enumerations

closure_selector

Enumerates options for defining if polygons are open or closed.

Description

The enumeration `closure_selector` describes options for if a polygon is open or closed. In a closed polygon the very first point (per ring) should be equal to the very last point. The specific closing property of a polygon type is defined by the closure metafunction. The closure metafunction defines a value, which is one of the values enumerated in the `closure_selector`

Synopsis

```
enum closure_selector {open = 0, closed = 1, closure_undertermined = -1};
```

Values

Value	Description
open	Rings are open: first point and last point are different, algorithms close them explicitly on the fly
closed	Rings are closed: first point and last point must be the same.
closure_undertermined	(Not yet implemented): algorithms first figure out if ring must be closed on the fly

Header

```
#include <boost/geometry/core/closure.hpp>
```

See also

[The closure metafunction](#)

order_selector

Enumerates options for the order of points within polygons.

Description

The enumeration `order_selector` describes options for the order of points within a polygon. Polygons can be ordered either clockwise or counterclockwise. The specific order of a polygon type is defined by the `point_order` metafunction. The `point_order` metafunction defines a value, which is one of the values enumerated in the `order_selector`

Synopsis

```
enum order_selector {clockwise = 1, counterclockwise = 2, order_undetermined = 0};
```

Values

Value	Description
<code>clockwise</code>	Points are ordered clockwise.
<code>counterclockwise</code>	Points are ordered counter clockwise.
<code>order_undetermined</code>	Points might be stored in any order, algorithms will determine it on the fly (not yet supported)

Header

```
#include <boost/geometry/core/point_order.hpp>
```

See also

[The point_order metafunction](#)

Exceptions

exception

Base exception class for Boost.Geometry algorithms.

Description

This class is never thrown. All exceptions thrown in Boost.Geometry are derived from `exception`, so it might be convenient to catch it.

Synopsis

```
class exception
    : public std::exception
{
    // ...
};
```

Header

```
#include <boost/geometry/core/exception.hpp>
```

centroid_exception

Centroid Exception.

Description

The `centroid_exception` is thrown if the free centroid function is called with geometries for which the centroid cannot be calculated. For example: a linestring without points, a polygon without points, an empty multi-geometry.

Synopsis

```
class centroid_exception
    : public exception
{
    // ...
};
```

Constructor(s)

Function	Description	Parameters
<code>centroid_exception()</code>		

Member Function(s)

Function	Description	Parameters	Returns
<code>char const * what()</code>			

Header

Either

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

Or

```
#include <boost/geometry/algorithms/centroid.hpp>
```

See also

- [the centroid function](#)

Iterators

closing_iterator

Iterator which iterates through a range, but adds first element at end of the range.

Synopsis

```
template<typename Range>
struct closing_iterator
{
    // ...
};
```

Template parameter(s)

Parameter	Description
typename Range	range on which this class is based on

Constructor(s)

Function	Description	Parameters
<code>closing_iterator< Range > or(Range & range)</code>	Constructor including the range it is based on.	Range &: <i>range</i> :
<code>closing_iterator< Range > or(Range & range, bool)</code>	Constructor to indicate the end of a range.	Range &: <i>range</i> : bool: :
<code>closing_iterator()</code>	Default constructor.	

Member Function(s)

Function	Description	Parameters	Returns
<code>closing_iterator< Range > & operator=(closing_iterator< Range > const & source)</code>		closing_iterator< Range > const &: <i>source</i> :	

Header

```
#include <boost/geometry/iterators/closing_iterator.hpp>
```

ever_circling_iterator

Iterator which ever circles through a range.

Synopsis

```
template<typename Iterator>
struct ever_circling_iterator
{
    // ...
};
```

Template parameter(s)

Parameter	Description
typename Iterator	iterator on which this class is based on

Constructor(s)

Function	Description	Parameters
<pre>ever_circling_iterator(Iterator begin, Iterator end, bool skip_first = false)</pre>		Iterator: <i>begin</i> : Iterator: <i>end</i> : bool: <i>skip_first</i> :
<pre>ever_circling_iterator(Iterator begin, Iterator end, Iterator start, bool skip_first = false)</pre>		Iterator: <i>begin</i> : Iterator: <i>end</i> : Iterator: <i>start</i> : bool: <i>skip_first</i> :

Member Function(s)

Function	Description	Parameters	Returns
<pre>void moveto(Iterator or it)</pre>		Iterator: <i>it</i> :	

Header

```
#include <boost/geometry/iterators/ever_circling_iterator.hpp>
```

Models

model::point

Basic point class, having coordinates defined in a neutral way.

Description

Defines a neutral point class, fulfilling the Point Concept. Library users can use this point class, or use their own point classes. This point class is used in most of the samples and tests of Boost.Geometry. This point class is used occasionally within the library, where a temporary point class is necessary.

Model of

Point Concept

Synopsis

```
template<typename CoordinateType, std::size_t DimensionCount, typename CoordinateSystem>
class model::point
{
    // ...
};
```

Template parameter(s)

Parameter	Description
typename CoordinateType	numerical type (int, double, ttmath, ...)
std::size_t DimensionCount	number of coordinates, usually 2 or 3
typename CoordinateSystem	coordinate system, for example cs::cartesian

Constructor(s)

Function	Description	Parameters
<code>point()</code>	Default constructor, no initialization.	
<pre>point(CoordinateType const & v0, CoordinateType const & v1 = 0, CoordinateType const & v2 = 0)</pre>	Constructor to set one, two or three values.	CoordinateType const &: v0: CoordinateType const &: v1: CoordinateType const &: v2:

Member Function(s)

Function	Description	Parameters	Returns
<pre>template<std::size_t K> CoordinateType const & get()</pre>	Get a coordinate.		the coordinate
<pre>template<std::size_t K> void set(CoordinateType const & value)</pre>	Set a coordinate.	CoordinateType const &: value: value to set	

Header

Either

```
#include <boost/geometry/geometries/geometries.hpp>
```

Or

```
#include <boost/geometry/geometries/point.hpp>
```

Examples

Declaration and use of the Boost.Geometry model::point, modelling the Point Concept

```
#include <iostream>
#include <boost/geometry.hpp>

namespace bg = boost::geometry;

int main()
{
    bg::model::point<double, 2, bg::cs::cartesian> point1;
    bg::model::point<double, 3, bg::cs::cartesian> point2(1.0, 2.0, 3.0); ❶
    point1.set<0>(1.0); ❷
    point1.set<1>(2.0);

    double x = point1.get<0>(); ❸
    double y = point1.get<1>();

    std::cout << x << ", " << y << std::endl;
    return 0;
}
```

- ❶ Construct, assigning three coordinates
- ❷ Set a coordinate. **Note:** prefer using `bg::set<0>(point1, 1.0);`
- ❸ Get a coordinate. **Note:** prefer using `x = bg::get<0>(point1);`

Output:

```
1, 2
```

Notes



Note

Coordinates are not initialized. If the constructor with parameters is not called and points are not assigned using set or assign then the coordinate values will contain garbage

model::d2::point_xy

2D point in Cartesian coordinate system

Model of

Point Concept

Synopsis

```
template<typename CoordinateType, typename CoordinateSystem>
class model::d2::point_xy
    : public model::point< CoordinateType, 2, CoordinateSystem >
{
    // ...
};
```

Template parameter(s)

Parameter	Default	Description
typename CoordinateType		numeric type, for example, double, float, int
typename CoordinateSystem	cs::cartesian	coordinate system, defaults to cs::cartesian

Constructor(s)

Function	Description	Parameters
<code>point_xy()</code>	Default constructor, does not initialize anything.	
<code>point_xy(CoordinateType const & x, CoordinateType const & y)</code>	Constructor with x/y values.	CoordinateType const &: x: CoordinateType const &: y:

Member Function(s)

Function	Description	Parameters	Returns
<code>CoordinateType const & x()</code>	Get x-value.		
<code>CoordinateType const & y()</code>	Get y-value.		
<code>void x(CoordinateType const & v)</code>	Set x-value.	CoordinateType const &: v:	
<code>void y(CoordinateType const & v)</code>	Set y-value.	CoordinateType const &: v:	

Header

```
#include <boost/geometry/geometries/point_xy.hpp>
```

Notes



Note

Coordinates are not initialized. If the constructor with parameters is not called and points are not assigned using set or assign then the coordinate values will contain garbage

model::linestring

A linestring (named so by OGC) is a collection (default a vector) of points.

Model of

Linestring Concept

Synopsis

```
template<typename Point, template< typename, typename > class Container, template< type,
name > class Allocator>
class model::linestring
{
    // ...
};
```

Template parameter(s)

Parameter	Default	Description
typename Point		Any type fulfilling a Point Concept
template< typename, typename > class Container	std::vector	container type, for example std::vector, std::deque
template< typename > class Allocator	std::allocator	container-allocator-type

Constructor(s)

Function	Description	Parameters
linestring()	Default constructor, creating an empty linestring.	
<pre>template<typename Iterator> linestring(Iterator begin, Iterator end)</pre>	Constructor with begin and end, filling the linestring.	Iterator: <i>begin</i> : Iterator: <i>end</i> :

Header

Either

```
#include <boost/geometry/geometries/geometries.hpp>
```

Or

```
#include <boost/geometry/geometries/linestring.hpp>
```

model::polygon

The polygon contains an outer ring and zero or more inner rings.

Model of

Polygon Concept

Synopsis

```
template<typename Point, bool ClockWise, bool Closed, template< typename, typename > class PointList,
template< typename, typename > class RingList, template< typename > class PointAlloc, template<
typename > class RingAlloc>
class model::polygon
{
    // ...
};
```

Template parameter(s)

Parameter	Default	Description
typename Point		point type
bool ClockWise	true	true for clockwise direction, false for CounterClockWise direction
bool Closed	true	true for closed polygons (last point == first point), false open points
template< typename, typename > class PointList	std::vector	container type for points, for example std::vector, std::list, std::deque
template< typename, typename > class RingList	std::vector	container type for inner rings, for example std::vector, std::list, std::deque
template< typename > class PointAlloc	std::allocator	container-allocator-type, for the points
template< typename > class RingAlloc	std::allocator	container-allocator-type, for the rings

Member Function(s)

Function	Description	Parameters	Returns
<code>ring_type const & outer()</code>			
<code>inner_container_type const & inneres()</code>			
<code>ring_type & outer()</code>			
<code>inner_container_type & inneres()</code>			
<code>void clear()</code>	Utility method, clears outer and inner rings.		

Header

Either

```
#include <boost/geometry/geometries/geometries.hpp>
```

Or

```
#include <boost/geometry/geometries/polygon.hpp>
```

model::multi_point

multi_point, a collection of points

Model of

MultiPoint Concept

Synopsis

```
template<typename Point, template< typename, typename > class Container, template< type-
name > class Allocator>
class model::multi_point
{
    // ...
};
```

Template parameter(s)

Parameter	Default	Description
typename Point		Any type fulfilling a Point Concept
template< typename, typename > class Container	std::vector	container type, for example std::vector, std::deque
template< typename > class Allocator	std::allocator	container-allocator-type

Constructor(s)

Function	Description	Parameters
<code>multi_point()</code>	Default constructor, creating an empty multi_point.	
<code>template<typename Iterator> multi_point(Iterator begin, Iterator end)</code>	Constructor with begin and end, filling the multi_point.	Iterator: <i>begin</i> : Iterator: <i>end</i> :

Header

```
#include <boost/geometry/multi/geometries/multi_point.hpp>
```

model::multi_linestring

multi_line, a collection of linestring

Description

Multi-linestring can be used to group lines belonging to each other, e.g. a highway (with interruptions)

Model of

MultiLineString Concept

Synopsis

```
template<typename LineString, template< typename, typename > class Container, template< type-  
name > class Allocator>  
class model::multi_linestring  
{  
    // ...  
};
```

Template parameter(s)

Parameter	Default	Description
typename LineString		
template< typename, typename > class Container	std::vector	
template< typename > class Allocator	std::allocator	

Header

```
#include <boost/geometry/multi/geometries/multi_linestring.hpp>
```

model::multi_polygon

multi_polygon, a collection of polygons

Description

Multi-polygon can be used to group polygons belonging to each other, e.g. Hawaii

Model of

MultiPolygon Concept

Synopsis

```
template<typename Polygon, template< typename, typename > class Container, template< type,
name > class Allocator>
class model::multi_polygon
{
    // ...
};
```

Template parameter(s)

Parameter	Default	Description
typename Polygon		
template< typename, typename > class Container	std::vector	
template< typename > class Allocator	std::allocator	

Header

```
#include <boost/geometry/multi/geometries/multi_polygon.hpp>
```

model::box

Class box: defines a box made of two describing points.

Description

Box is always described by a min_corner() and a max_corner() point. If another rectangle is used, use linear_ring or polygon.

Synopsis

```
template<typename Point>
class model::box
{
    // ...
};
```

Template parameter(s)

Parameter	Description
typename Point	point type. The box takes a point type as template parameter. The point type can be any point type. It can be 2D but can also be 3D or more dimensional. The box can also take a latlong point type as template parameter.

Constructor(s)

Function	Description	Parameters
box()		
box(Point const &min_corner, Point const &max_corner)	Constructor taking the minimum corner point and the maximum corner point.	Point const &: min_corner: Point const &: max_corner:

Member Function(s)

Function	Description	Parameters	Returns
Point const &min_corner()			
Point const &max_corner()			
Point &min_corner()			
Point &max_corner()			

Header

Either

```
#include <boost/geometry/geometries/geometries.hpp>
```

Or

```
#include <boost/geometry/geometries/box.hpp>
```

model::ring

A ring (aka linear ring) is a closed line which should not be selfintersecting.

Model of

Ring Concept

Synopsis

```
template<typename Point, bool ClockWise, bool Closed, template< typename, typename > class Container,
template< typename > class Allocator>
class model::ring
{
    // ...
};
```

Template parameter(s)

Parameter	Default	Description
typename Point		point type
bool ClockWise	true	true for clockwise direction, false for CounterClockWise direction
bool Closed	true	true for closed polygons (last point == first point), false open points
template< typename, typename > class Container	std::vector	container type, for example std::vector, std::deque
template< typename > class Allocator	std::allocator	container-allocator-type

Constructor(s)

Function	Description	Parameters
ring()	Default constructor, creating an empty ring.	
<pre>template<typename Iterator> ring(Iterator begin, Iterator end)</pre>	Constructor with begin and end, filling the ring.	Iterator: <i>begin</i> : Iterator: <i>end</i> :

Header

Either

```
#include <boost/geometry/geometries/geometries.hpp>
```

Or

```
#include <boost/geometry/geometries/ring.hpp>
```

model::segment

Class segment: small class containing two points.

Description

From Wikipedia: In geometry, a line segment is a part of a line that is bounded by two distinct end points, and contains every point on the line between its end points.

Synopsis

```
template<typename Point>
class model::segment
{
    // ...
};
```

Template parameter(s)

Parameter	Description
typename Point	

Constructor(s)

Function	Description	Parameters
<code>segment()</code>		
<code>segment(Point const & p1, Point const & p2)</code>		Point const &: p1: Point const &: p2:

Header

Either

```
#include <boost/geometry/geometries/geometries.hpp>
```

Or

```
#include <boost/geometry/geometries/segment.hpp>
```

model::referring_segment

Class segment: small class containing two (templated) point references.

Description

From Wikipedia: In geometry, a line segment is a part of a line that is bounded by two distinct end points, and contains every point on the line between its end points.

Synopsis

```
template<typename ConstOrNonConstPoint>
class model::referring_segment
{
    // ...
};
```

Template parameter(s)

Parameter	Description
typename ConstOrNonConstPoint	point type of the segment, maybe a point or a const point

Constructor(s)

Function	Description	Parameters
<pre>referring_segment(point_type & p1, point_type & p2)</pre>		<p>point_type &: p1:</p> <p>point_type &: p2:</p>

Header

Either

```
#include <boost/geometry/geometries/geometries.hpp>
```

Or

```
#include <boost/geometry/geometries/segment.hpp>
```

Strategies

strategy::distance::pythagoras

Strategy to calculate the distance between two points.

Synopsis

```
template<typename Point1, typename Point2, typename CalculationType>
class strategy::distance::pythagoras
{
    // ...
};
```

Template parameter(s)

Parameter	Default	Description
typename Point1		point type
typename Point2	Point1	second point type
typename CalculationType	void	numeric type for calculation (e.g. high precision); if void then it is extracted automatically from the coordinate type and (if necessary) promoted to floating point

Member Function(s)

Function	Description	Parameters	Returns
<pre>calculation_type applyDistance(const Point1& p1, const Point2& p2)</pre>	applies the distance calculation using pythagoras	Point1 const &: p1 : first point Point2 const &: p2 : second point	the calculated distance (including taking the square root)

Header

```
#include <boost/geometry/strategies/cartesian/distance_pythagoras.hpp>
```

Notes**Note**

Can be used for points with two, three or more dimensions

See also

[distance \(with strategy\)](#)

strategy::distance::haversine

Distance calculation for spherical coordinates on a perfect sphere using haversine.

Synopsis

```
template<typename Point1, typename Point2, typename CalculationType>
class strategy::distance::haversine
{
    // ...
};
```

Template parameter(s)

Parameter	Default	Description
typename Point1		point type
typename Point2	Point1	second point type
typename CalculationType	void	numeric type for calculation (e.g. high precision); if void then it is extracted automatically from the coordinate type and (if necessary) promoted to floating point

Constructor(s)

Function	Description	Parameters
<pre>haversine(calculation_type const & radius, Point1 const & p1, Point2 const & p2)</pre>	Constructor.	calculation_type const &: <i>radius</i> : radius of the sphere, defaults to 1.0 for the unit sphere

Member Function(s)

Function	Description	Parameters	Returns
<pre>calculation_type apply(Point1 const & p1, Point2 const & p2)</pre>	applies the distance calculation	Point1 const &: <i>p1</i> : first point Point2 const &: <i>p2</i> : second point	the calculated distance (including multiplying with radius)
<pre>calculation_type radius()</pre>	access to radius value		the radius

Header

```
#include <boost/geometry/strategies/spherical/distance_haversine.hpp>
```

See also

[distance \(with strategy\)](#)

strategy::distance::projected_point

Strategy for distance point to segment.

Description

Calculates distance using projected-point method, and (optionally) Pythagoras

Synopsis

```
template<typename Point, typename PointOfSegment, typename CalculationType, typename Strategy>
class strategy::distance::projected_point
{
    // ...
};
```

Template parameter(s)

Parameter	Default	Description
typename Point		Any type fulfilling a Point Concept
typename PointOfSegment	Point	segment point type
typename CalculationType	void	numeric type for calculation (e.g. high precision); if void then it is extracted automatically from the coordinate type and (if necessary) promoted to floating point
typename Strategy	pythagoras<Point, PointOfSegment, CalculationType>	underlying point-point distance strategy

Member Function(s)

Function	Description	Parameters	Returns
<pre>calculation_type apply(Point const & p, PointOfSegment const & p1, PointOfSegment const & p2) const;</pre>		Point const &: <i>p</i>: PointOfSegment const &: <i>p1</i>: PointOfSegment const &: <i>p2</i>:	

Header

```
#include <boost/geometry/strategies/cartesian/distance_projected_point.hpp>
```

See also

[distance \(with strategy\)](#)

strategy::distance::cross_track

Strategy functor for distance point to segment calculation.

Description

Class which calculates the distance of a point to a segment, using latlong points

Synopsis

```
template<typename Point, typename PointOfSegment, typename CalculationType, typename Strategy>
class strategy::distance::cross_track
{
    // ...
};
```

Template parameter(s)

Parameter	Default	Description
typename Point		point type
typename PointOfSegment	Point	segment point type
typename CalculationType	void	numeric type for calculation (e.g. high precision); if void then it is extracted automatically from the coordinate type and (if necessary) promoted to floating point
typename Strategy	typename services::default_strategy<point_tag, Point>::type	underlying point-point distance strategy, defaults to haversine

Constructor(s)

Function	Description	Parameters
<code>cross_track()</code>		
<code>cross_track(return_type const & r)</code>		return_type const &: r:
<code>cross_track(Strategy const & s)</code>		Strategy const &: s:

Member Function(s)

Function	Description	Parameters	Returns
<code>return_type apply(Point const & p, PointOfSegment const & sp1, PointOfSegment const & sp2)</code>		Point const &: p: PointOfSegment const &: sp1: PointOfSegment const &: sp2:	
<code>return_type radius()</code>			

Header

```
#include <boost/geometry/strategies/spherical/distance_cross_track.hpp>
```

See also

[distance \(with strategy\)](#)

strategy::area::surveyor

Area calculation for cartesian points.

Description

Calculates area using the Surveyor's formula, a well-known triangulation algorithm

Synopsis

```
template<typename PointOfSegment, typename CalculationType>
class strategy::area::surveyor
{
    // ...
};
```

Template parameter(s)

Parameter	Default	Description
typename PointOfSegment		segment point type
typename CalculationType	void	numeric type for calculation (e.g. high precision); if void then it is extracted automatically from the coordinate type and (if necessary) promoted to floating point

Member Function(s)

Function	Description	Parameters	Returns
<pre>void apply(PointOfSegment const & p1, PointOfSegment const & p2, summation & state)</pre>		PointOfSegment const &: p1: PointOfSegment const &: p2: summation &: state:	
<pre>return_type result(summation const & state)</pre>		summation const &: state:	

Header

```
#include <boost/geometry/strategies/cartesian/area_surveyor.hpp>
```

See also

[area \(with strategy\)](#)

strategy::area::huiller

Area calculation by spherical excess / Huiller's formula.

Synopsis

```
template<typename PointOfSegment, typename CalculationType>
class strategy::area::huiller
{
    // ...
};
```

Template parameter(s)

Parameter	Default	Description
typename PointOfSegment		point type of segments of rings/polygons
typename CalculationType	void	numeric type for calculation (e.g. high precision); if void then it is extracted automatically from the coordinate type and (if necessary) promoted to floating point

Constructor(s)

Function	Description	Parameters
<pre>huiller(calculation_type radius = 1.0)</pre>		calculation_type : <i>radius</i> :

Member Function(s)

Function	Description	Parameters	Returns
<pre>void apply(PointOfSegment const & p1, PointOfSegment const & p2, excess_sum & state)</pre>		PointOfSegment const &: p1: PointOfSegment const &: p2: excess_sum &: state:	
<pre>return_type result(excess_sum const & state)</pre>		excess_sum const &: state:	

Header

```
#include <boost/geometry/strategies/spherical/area_huiller.hpp>
```

Example

Calculate the area of a polygon

```
#include <iostream>

#include <boost/geometry.hpp>
#include <boost/geometry/geometries/point_xy.hpp>
#include <boost/geometry/geometries/polygon.hpp>
#include <boost/geometry/io/wkt/wkt.hpp>

namespace bg = boost::geometry; ❶

int main()
{
    // Calculate the area of a cartesian polygon
    bg::model::polygon<bg::model::d2::point_xy<double> > poly;
    bg::read_wkt("POLYGON((0 0,0 7,4 2,2 0,0 0))", poly);
    double area = bg::area(poly);
    std::cout << "Area: " << area << std::endl;

    // Calculate the area of a spherical polygon (for latitude: 0 at equator)
    bg::model::polygon<bg::model::point<float, 2, bg::cs::spherical_equatorial<bg::degree> > > sph_poly;
    bg::read_wkt("POLYGON((0 0,0 45,45 0,0 0))", sph_poly);
    area = bg::area(sph_poly);
    std::cout << "Area: " << area << std::endl;

    return 0;
}
```

❶ Convenient namespace alias

Output:

```
Area: 16
Area: 0.339837
```

See also

[area \(with strategy\)](#)

strategy::centroid::average

Centroid calculation taking average of points.

Synopsis

```
template<typename PointCentroid, typename Point>
class strategy::centroid::average
{
    // ...
};
```

Template parameter(s)

Parameter	Default	Description
typename PointCentroid		
typename Point	PointCentroid	

Member Function(s)

Function	Description	Parameters	Returns
<pre>void apply(Point const & p, sum & state)</pre>		Point const &: <i>p</i>: sum &: <i>state</i>:	
<pre>void result(sum const & state, PointCentroid & centroid)</pre>		sum const &: <i>state</i>: PointCentroid &: <i>centroid</i>:	

Header

Either

```
#include <boost/geometry/multi/multi.hpp>
```

Or

```
#include <boost/geometry/multi/strategies/cartesian/centroid_average.hpp>
```

strategy::centroid::bashein_detmer

Centroid calculation using algorithm Bashein / Detmer.

Description

Calculates centroid using triangulation method published by Bashein / Detmer

Statements:

With holes:

Statements:

Synopsis

```
template<typename Point, typename PointOfSegment, typename CalculationType>
class strategy::centroid::bashein_detmer
{
    // ...
};
```

Template parameter(s)

Parameter	Default	Description
typename Point		point type of centroid to calculate
typename PointOfSegment	Point	point type of segments, defaults to Point
typename CalculationType	void	

Member Function(s)

Function	Description	Parameters	Returns
<pre>void apply(PointOfSegment const & p1, PointOfSegment const & p2, sums & state)</pre>		PointOfSegment const &: p1: PointOfSegment const &: p2: sums &: state:	
<pre>bool result(sums const & state, Point & centroid)</pre>		sums const &: state: Point &: centroid:	

Header

```
#include <boost/geometry/strategies/cartesian/centroid_bashein_detmer.hpp>
```

See also

[centroid \(with strategy\)](#)

strategy::convex_hull::graham_andrew

Graham scan strategy to calculate convex hull.

Synopsis

```
template<typename InputGeometry, typename OutputPoint>
class strategy::convex_hull::graham_andrew
{
    // ...
};
```

Template parameter(s)

Parameter	Description
typename InputGeometry	
typename OutputPoint	

Member Function(s)

Function	Description	Parameters	Returns
<pre>void apply(InputGeometry const & geometry, partitions & state)</pre>		InputGeometry const &: <i>geometry</i> :	
<pre>template<typename OutputIterator> void result(partitions const & state, OutputIterator out, bool clockwise)</pre>		partitions const &: <i>state</i> : OutputIterator: <i>out</i> : bool: <i>clockwise</i> :	

Header

```
#include <boost/geometry/strategies/agnostic/hull_graham_andrew.hpp>
```

strategy::side::side_by_triangle

Check at which side of a segment a point lies: left of segment (> 0), right of segment (< 0), on segment (0)

Synopsis

```
template<typename CalculationType>
class strategy::side::side_by_triangle
{
    // ...
};
```

Template parameter(s)

Parameter	Default	Description
typename CalculationType	void	numeric type for calculation (e.g. high precision); if void then it is extracted automatically from the coordinate type and (if necessary) promoted to floating point

Member Function(s)

Function	Description	Parameters	Returns
<pre>template<typename P1, typename P2, typename P> int apply(P1 const & p1, P2 const & p2, P const & p)</pre>		P1 const &: <i>p1</i> : P2 const &: <i>p2</i> : P const &: <i>p</i> :	

Header

```
#include <boost/geometry/strategies/cartesian/side_by_triangle.hpp>
```

strategy::side::side_by_cross_track

Check at which side of a Great Circle segment a point lies left of segment (> 0), right of segment (< 0), on segment (0)

Synopsis

```
template<typename CalculationType>
class strategy::side::side_by_cross_track
{
    // ...
};
```

Template parameter(s)

Parameter	Default	Description
typename CalculationType	void	numeric type for calculation (e.g. high precision); if void then it is extracted automatically from the coordinate type and (if necessary) promoted to floating point

Member Function(s)

Function	Description	Parameters	Returns
<pre>template<typename P1, typename P2, typename P> int apply(const P1& p1, const P2& p2, const P& p)</pre>		P1 const &: <i>p1</i>: P2 const &: <i>p2</i>: P const &: <i>p</i>:	

Header

```
#include <boost/geometry/strategies/spherical/side_by_cross_track.hpp>
```

strategy::side::spherical_side_formula

Check at which side of a Great Circle segment a point lies left of segment (> 0), right of segment (< 0), on segment (0)

Synopsis

```
template<typename CalculationType>
class strategy::side::spherical_side_formula
{
    // ...
};
```

Template parameter(s)

Parameter	Default	Description
typename CalculationType	void	numeric type for calculation (e.g. high precision); if void then it is extracted automatically from the coordinate type and (if necessary) promoted to floating point

Member Function(s)

Function	Description	Parameters	Returns
<pre>template<typename name P1, typename name P2, typename P> int ap(P1 const &p1, P2 const &p2, P const &p)</pre>		P1 const &: <i>p1</i>: P2 const &: <i>p2</i>: P const &: <i>p</i>:	

Header

```
#include <boost/geometry/strategies/spherical/ssf.hpp>
```

strategy::simplify::douglas_peucker

Implements the simplify algorithm.

Description

The douglas_peucker strategy simplifies a linestring, ring or vector of points using the well-known Douglas-Peucker algorithm. For the algorithm, see for example:

Synopsis

```
template<typename Point, typename PointDistanceStrategy>
class strategy::simplify::douglas_peucker
{
    // ...
};
```

Template parameter(s)

Parameter	Description
typename Point	the point type
typename PointDistanceStrategy	point-segment distance strategy to be used

Member Function(s)

Function	Description	Parameters	Returns
<pre>template<typename Range, typename OutputIterator> OutputIterator apply(const Range & range, OutputIterator out, double max_distance)</pre>		Range const &: <i>range</i> : OutputIterator : <i>out</i> : double : <i>max_distance</i> :	

Header

```
#include <boost/geometry/strategies/agnostic/simplify_douglas_peucker.hpp>
```

strategy::transform::inverse_transformer

Transformation strategy to do an inverse transformation in Cartesian system.

Synopsis

```
template<typename P1, typename P2>
class strategy::transform::inverse_transformer
    : public strategy::transform::ublas_transformer< P1, P2, dimension< P1 >::type::value, dimension< P2 >::type::value >
{
    // ...
};
```

Template parameter(s)

Parameter	Description
typename P1	first point type
typename P2	second point type

Constructor(s)

Function	Description	Parameters
<pre>template<typename Transformer> inverse_transformer(Transformer const & input)</pre>		Transformer const &: <i>input</i> :

Header

```
#include <boost/geometry/strategies/transform/inverse_transformer.hpp>
```

strategy::transform::map_transformer

Transformation strategy to do map from one to another Cartesian system.

Synopsis

```
template<typename P1, typename P2, bool Mirror, bool SameScale, std::size_t Dimension1, std::size_t Dimension2>
class strategy::transform::map_transformer
    : public strategy::transform::ublas_transformer< P1, P2, Dimension1, Dimension2 >
{
    // ...
};
```

Template parameter(s)

Parameter	Default	Description
typename P1		first point type
typename P2		second point type
bool Mirror	false	if true map is mirrored upside-down (in most cases pixels are from top to bottom, while map is from bottom to top)
bool SameScale	true	
std::size_t Dimension1	dimension<P1>::type::value	
std::size_t Dimension2	dimension<P2>::type::value	

Constructor(s)

Function	Description	Parameters
<pre>template<typename B, type_ name D> map_trans_ fomer(B const &box, D const &width, D const &height)</pre>		B const &: <i>box</i> : D const &: <i>width</i> : D const &: <i>height</i> :
<pre>template<typename W, type_ name D> map_trans_ fomer(W const &wx1, W const &wx2, W const &wy1, W const &wy2, D const &width, D const &height)</pre>		W const &: <i>wx1</i> : W const &: <i>wy1</i> : W const &: <i>wx2</i> : W const &: <i>wy2</i> : D const &: <i>width</i> : D const &: <i>height</i> :

Header

```
#include <boost/geometry/strategies/transform/map_transformer.hpp>
```

strategy::transform::rotate_transformer

Strategy of rotate transformation in Cartesian system.

Description

Rotate rotates a geometry of specified angle about a fixed point (e.g. origin).

Synopsis

```
template<typename P1, typename P2, typename DegreeOrRadian>
class strategy::transform::rotate_transformer
{
    // ...
};
```

Template parameter(s)

Parameter	Description
typename P1	first point type
typename P2	second point type
typename DegreeOrRadian	degree/or/radian, type of rotation angle specification

Constructor(s)

Function	Description	Parameters
<pre>rotate_transformer(angle_type const & angle)</pre>		angle_type const &: angle:

Header

```
#include <boost/geometry/strategies/transform/matrix_transformers.hpp>
```

strategy::transform::scale_transformer

Strategy of scale transformation in Cartesian system.

Description

Scale scales a geometry up or down in all its dimensions.

Synopsis

```
template<typename P1, typename P2, std::size_t Dimension1, std::size_t Dimension2>
class strategy::transform::scale_transformer
{
    // ...
};
```

Template parameter(s)

Parameter	Default	Description
typename P1		first point type
typename P2	P1	second point type
std::size_t Dimension1	geometry::dimension<P1>::type::value	number of dimensions to transform to second point
std::size_t Dimension2	geometry::dimension<P2>::type::value	

Header

```
#include <boost/geometry/strategies/transform/matrix_transformers.hpp>
```

strategy::transform::translate_transformer

Strategy of translate transformation in Cartesian system.

Description

Translate moves a geometry a fixed distance in 2 or 3 dimensions.

Synopsis

```
template<typename P1, typename P2, std::size_t Dimension1, std::size_t Dimension2>
class strategy::transform::translate_transformer
{
    // ...
};
```

Template parameter(s)

Parameter	Default	Description
typename P1		first point type
typename P2		second point type
std::size_t Dimension1	geometry::dimension<P1>::type::value	number of dimensions to transform to second point
std::size_t Dimension2	geometry::dimension<P2>::type::value	

Header

```
#include <boost/geometry/strategies/transform/matrix_transformers.hpp>
```

strategy::transform::ublas_transformer

Affine transformation strategy in Cartesian system.

Description

The strategy serves as a generic definition of affine transformation matrix and procedure of application it to given point.

Synopsis

```
template<typename P1, typename P2, std::size_t Dimension1, std::size_t Dimension2>
class strategy::transform::ublas_transformer
{
    // ...
};
```

Template parameter(s)

Parameter	Description
typename P1	first point type (source)
typename P2	second point type (target)
std::size_t Dimension1	number of dimensions to transform to second point
std::size_t Dimension2	

Header

```
#include <boost/geometry/strategies/transform/matrix_transformers.hpp>
```

strategy::within::winding

Within detection using winding rule.

Synopsis

```
template<typename Point, typename PointOfSegment, typename CalculationType>
class strategy::within::winding
{
    // ...
};
```

Template parameter(s)

Parameter	Default	Description
typename Point		Any type fulfilling a Point Concept
typename PointOfSegment	Point	segment point type
typename CalculationType	void	numeric type for calculation (e.g. high precision); if void then it is extracted automatically from the coordinate type and (if necessary) promoted to floating point

Member Function(s)

Function	Description	Parameters	Returns
<pre>bool apply(Point const & point, PointOfSegment const & s1, PointOfSegment const & s2, counter & state)</pre>		Point const &: point: PointOfSegment const &: s1: PointOfSegment const &: s2: counter &: state:	
<pre>int result(counter const & state)</pre>		counter const &: state:	

Header

```
#include <boost/geometry/strategies/agnostic/point_in_poly_winding.hpp>
```

See also

within (with strategy)

strategy::within::franklin

Within detection using cross counting.

Synopsis

```
template<typename Point, typename PointOfSegment, typename CalculationType>
class strategy::within::franklin
{
    // ...
};
```

Template parameter(s)

Parameter	Default	Description
typename Point		Any type fulfilling a Point Concept
typename PointOfSegment	Point	segment point type
typename CalculationType	void	numeric type for calculation (e.g. high precision); if void then it is extracted automatically from the coordinate type and (if necessary) promoted to floating point

Member Function(s)

Function	Description	Parameters	Returns
<pre>bool apply(Point const & point, PointOfSegment const & seg1, PointOfSegment const & seg2, crossings & state)</pre>		Point const &: <i>point</i>: PointOfSegment const &: <i>seg1</i>: PointOfSegment const &: <i>seg2</i>: crossings &: <i>state</i>:	
<pre>int result(crossings const & state)</pre>		crossings const &: <i>state</i>:	

Header

```
#include <boost/geometry/strategies/cartesian/point_in_poly_franklin.hpp>
```

See also

[within](#) (with strategy)

strategy::within::crossings_multiply

Within detection using cross counting,.

Synopsis

```
template<typename Point, typename PointOfSegment, typename CalculationType>
class strategy::within::crossings_multiply
{
    // ...
};
```

Template parameter(s)

Parameter	Default	Description
typename Point		Any type fulfilling a Point Concept
typename PointOfSegment	Point	segment point type
typename CalculationType	void	numeric type for calculation (e.g. high precision); if void then it is extracted automatically from the coordinate type and (if necessary) promoted to floating point

Member Function(s)

Function	Description	Parameters	Returns
<pre>bool apply(Point const & point, PointOfSegment const & seg1, PointOfSegment const & seg2, flags & state)</pre>		Point const &: point: PointOfSegment const &: seg1: PointOfSegment const &: seg2: flags &: state:	
<pre>int result(const flags & state)</pre>		flags const &: state:	

Header

```
#include <boost/geometry/strategies/cartesian/point_in_poly_crossings_multiply.hpp>
```

Views

box_view

Makes a box behave like a ring or a range.

Description

Adapts a box to the Boost.Range concept, enabling the user to iterating box corners. The box_view is registered as a Ring Concept

Model of

Ring Concept

Synopsis

```
template<typename Box, bool Clockwise>
struct box_view
{
    // ...
};
```

Template parameter(s)

Parameter	Default	Description
typename Box		A type fulfilling the Box Concept
bool Clockwise	true	If true, walks in clockwise direction, otherwise it walks in counterclockwise direction

Constructor(s)

Function	Description	Parameters
<code>box_view(Box const & box)</code>	Constructor accepting the box to adapt.	Box const &: <i>box</i> :

Header

Either

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

Or

```
#include <boost/geometry/views/box_view.hpp>
```

Complexity

Compile time

Example

Shows usage of the Boost.Range compatible view on a box

```
#include <iostream>

#include <boost/geometry.hpp>

int main()
{
    typedef boost::geometry::model::box
        <
            boost::geometry::model::point<double, 2, boost::geometry::cs::cartesian>
        > box_type;

    // Define the Boost.Range compatible type:
    typedef boost::geometry::box_view<box_type> box_view;

    box_type box;
    boost::geometry::assign_values(box, 0, 0, 4, 4);

    box_view view(box);

    // Iterating in clockwise direction over the points of this box
    for (boost::range_iterator<box_view const>::type it = boost::begin(view);
         it != boost::end(view); ++it)
    {
        std::cout << " " << boost::geometry::dsv(*it);
    }
    std::cout << std::endl;

    // Note that a box_view is tagged as a ring, so supports area etc.
    std::cout << "Area: " << boost::geometry::area(view) << std::endl;

    return 0;
}
```

Output:

```
(0, 0) (0, 4) (4, 4) (4, 0) (0, 0)
Area: 16
```

segment_view

Makes a segment behave like a linestring or a range.

Description

Adapts a segment to the Boost.Range concept, enabling the user to iterate the two segment points. The segment_view is registered as a LineString Concept

Model of

LineString Concept

Synopsis

```
template<typename Segment>
struct segment_view
{
    // ...
};
```

Template parameter(s)

Parameter	Description
typename Segment	A type fulfilling the Segment Concept

Constructor(s)

Function	Description	Parameters
<pre>segment_view(Segment const & segment)</pre>	Constructor accepting the segment to adapt.	Segment const &: <i>segment</i>:

Header

Either

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

Or

```
#include <boost/geometry/views/segment_view.hpp>
```

Complexity

Compile time

Example

Shows usage of the Boost.Range compatible view on a box

```

#include <iostream>

#include <boost/geometry.hpp>

int main()
{
    typedef boost::geometry::model::segment
        <
            boost::geometry::model::point<double, 2, boost::geometry::cs::cartesian>
        > segment_type;

    typedef boost::geometry::segment_view<segment_type> segment_view;

    segment_type segment;
    boost::geometry::assign_values(segment, 0, 0, 1, 1);

    segment_view view(segment);

    // Iterating over the points of this segment
    for (boost::range_iterator<segment_view const>::type it = boost::begin(view);
        it != boost::end(view); ++it)
    {
        std::cout << " " << boost::geometry::dsv(*it);
    }
    std::cout << std::endl;

    // Note that a segment_view is tagged as a linestring, so supports length etc.
    std::cout << "Length: " << boost::geometry::length(view) << std::endl;

    return 0;
}

```

Output:

```

(0, 0) (0, 4) (4, 4) (4, 0) (0, 0)
Area: 16

```

closeable_view

View on a range, either closing it or leaving it as it is.

Description

The `closeable_view` is used internally by the library to handle all rings, either closed or open, the same way. The default method is closed, all algorithms process rings as if they are closed. Therefore, if they are opened, a view is created which closes them. The `closeable_view` might be used by library users, but its main purpose is internally.

Synopsis

```

template<typename Range, closure_selector Close>
struct closeable_view
{
    // ...
};

```

Template parameter(s)

Parameter	Description
typename Range	Original range
closure_selector Close	Specifies if it the range is closed, if so, nothing will happen. If it is open, it will iterate the first point after the last point.

Header

```
#include <boost/geometry/views/closeable_view.hpp>
```

reversible_view

View on a range, reversing direction if necessary.

Synopsis

```
template<typename Range, iterate_direction Direction>
struct reversible_view
{
    // ...
};
```

Template parameter(s)

Parameter	Description
typename Range	original range
iterate_direction Direction	direction of iteration

Header

```
#include <boost/geometry/views/reversible_view.hpp>
```

identity_view

View on a range, not modifying anything.

Synopsis

```
template<typename Range>
struct identity_view
{
    // ...
};
```

Template parameter(s)

Parameter	Description
typename Range	original range

Constructor(s)

Function	Description	Parameters
<code>identity_view(Range & r)</code>		Range &: <i>r</i>:

Member Function(s)

Function	Description	Parameters	Returns
<code>const_iterator begin()</code>			
<code>const_iterator end()</code>			
<code>iterator begin()</code>			
<code>iterator end()</code>			

Header

```
#include <boost/geometry/views/identity_view.hpp>
```

Release Notes

Boost 1.49

Breaking changes

- `point_xy` was accidentally included in one of the headerfiles. If the `point_xy` class is used, it should be included explicitly now.

Bugfixes

- bugfix: distance for multi-geometries ignored specified distance strategy. Fixed
- bugfix: difference for polygon/multi_polygon (reported 2011/10/24 on GGL-list)
- bugfix: raise exception for calculation of distances of multi-geometrie(s) where one of them is empty
- bugfix: multi DSV did not correctly use settings, fixed
- bugfix: self-intersections could sometimes be missed (introduced in 1.48), fixed
- bugfix: convex hull crashed on empty range (e.g. empty multi point), fixed
- bugfix: area/centroid/side/intersection/distance did not work for "int" type filled with large (but not overflowing) integers. Fixed.
- bugfix: disjoint/intersect did not work for degenerate linestrings. Fixed.
- bugfix: covered_by did not compile for a ring. Fixed.

Solved tickets

- [6019](#) convex_hull / area, fixed.
- [6021](#) convex_hull / append (multipoint), fixed.
- [6028](#) Documentation: closure, fixed.
- [6178](#) Missing headerfile, fixed.

Additional functionality

- support for line/polygon intersections and differences
- support for convert of segment/box of different point types
- support for append for multi point
- the scalar function distance now throws an `empty_input_exception` on empty input

Documentation

- updated support status in several algorithms
- updated conformance to OGC or std
- other updates and fixes

Internal changes

- updates in specializations/not_implemented for distance/convert/assign/area/with/covered_by
- move of wkt/dsv to io folder, making domains redundant

- warnings: strategy concepts assigned to zero to avoid clang warnings (patched by Vishnu)
- warnings: there were several unused parameters, for which gcc/clang warned (patched by Christophe)

Boost 1.48

Bugfixes

- Robustness issue, in some circumstances the union failed to output. Fixed.
- Robustness issue, in some circumstances the calculated intersection point was outside the segment. Fixed.
- Concept issue, cartesian intersect didn't understand segments other than the provided one. Fixed.
- Sometimes self-intersections in linestrings were missed. Fixed.
- The fusion coordinate system was not registered correctly. Fixed.

Solved tickets

- [5726](#) Segment intersection algorithm still assumes 'first', 'second' members
- [5744](#) Mistake in fusion adapt example
- [5748](#) Needed to include <boost/foreach.hpp>
- [5954](#) distance_pythagoras skips sqrt() step

Improvements on algorithms

- Checking self-intersections is now not done automatically, this can blast performance.
- Besides that, checking self-intersections is made faster.
- Intersections now avoid outputting duplicate points. So they output the minimal set.

Additional algorithms

- covered_by: within is defined as "within, not on boundary". covered_by is "within or on boundary"

Additional functionality

- within: strategies can now be specified for within<point, box> and within<box, box>
- convert: a much broader range of conversions is supported
- assign: idem, (currently partly) synonym for convert (but reversed arguments)

Additional coordinate types

- Basic (limited) support for Boost.Rational

Boost 1.47

Original release

About this documentation

Within the Boost community there are several styles of documenting. Most libraries nowadays are using QuickBook, the WikiWiki style documentation.

Boost.Geometry started with Doxygen, and during review it was decided to go to QuickBook. However, it was convenient to keep Doxygen also there: it does a good job of connecting descriptions to function and class declarations.

Doxygen is able to generate XML (besides the normal HTML output), containing all documentation.

So the challenge was to translate the XML generated by doxygen to QuickBook. At least, translate parts of it. Boost contains currently two tools using XSLT to go from Doxygen-XML to BoostBook, or to QuickBook. These tools are used within Boost.Random and Boost.Asio (and maybe more). However, this XSLT process was quite hard, did not deliver (yet) the wished results, and we are all C++ programmers. So another tool was born, this time in C++ using RapidXML, going from Doxygen-XML to QuickBook with the ability to mix both.

The chain

The process is as following:

1. call doxygen to go from C++ to XML
2. call *doxygen_xml2qbk* to go from XML to QuickBook
3. call bjam to from QuickBook to HTML
 - a. bjam translates QuickBook to BoostBook
 - b. bjam then translates from BoostBook to DocBook
 - c. bjam then translates from DocBook to HTML

This chain is currently called by "make_qbk.py", a Python script which calls the chain above in the right order. Python ensures that the chain can be handled in both Windows and Linux environments (it is probably possible to call all parts with bjam too).

The reference matrix

There reference matrix is the only thing written in BoostBook. It is an XML file with an overview of all Boost.Geometry functionality. Presenting it like this is not possible within QuickBook, therefore BoostBook XML is used here. It is included by the QuickBook code. The Boost.Asio documentation contains a similar reference matrix.

Mixing QuickBook into C++ code

With Doxygen it is possible to define aliases. Specificly for *doxygen_xml2qbk*, the alias `\qbk{...}` was defined. This alias `\qbk{...}` add some XML-tags around the text inside the alias, such that that included part is recognizable by the converter.

So the C++ code might look like this:

```
/*!
 \brief Some explanation
 \ingroup some_group
 \details Some details
 \tparam Geometry Description of the template parameter
 \param geometry Description of the variable

 \qbk{ [include reference/more_documentation.qbk] }
 */
```


First you see normal Doxygen comments. The last line uses the alias `\qbk{...}` to include a QuickBook file. So most of the documentation can be written in that QuickBook file: behaviour, complexity, examples, related pages, etc.

In the example above a QuickBook include statement is used. But any QuickBook code can be used, the QuickBook code does not have to be stored in a separate file. Two more samples:

```
/*!  
...  
\qbk{  
[heading Example]  
[area_with_strategy]  
[area_with_strategy_output]  
  
[heading Available Strategies]  
[link geometry.reference.strategies.strategy_area_surveyor Surveyor (cartesian)]  
}  
*/
```

In this example pieces of QuickBook are included, two headers, two examples (this is the QuickBook way - the examples are defined elsewhere), and a link.

QuickBook within C++ issues

There are two issues: the comma and the asterisk. If within the `\qbk` alias a comma is used, it is recognized by Doxygen as another parameter, and therefore will not deliver the correct results, or result into errors. This is easily solvable by escaping comma's (by putting a backslash directly before the comma, `\,`). It within the `\qbk` alias an asterisk is used on the first line, it is interpreted by Doxygen as well. This asterisk can be escaped as well, and this time it is `doxygen_qbk2xml` which handles this escape and translates it back into an asterisk.

Overloads

Boost.Geometry contains a lot of overloads, two functions with the same name and, for example, a different number of parameters. Or, as another example, a const and a non-const version. They can be marked specifically to the `doxygen_xml2qbk` tool with the `\qbk` alias, by adding a specific description for the overload. So, for example, `\qbk{distinguish,with strategy}` will result in another page where the text "with strategy" is added, and it is processed as `"_with_strategy"` within the QuickBook section name.

Overloads and sharing documentation

With overloads, part of the documentation must be shared, and other part must not. The descriptions are often the same. But the examples are usually not. So it is a balance between sharing documentation, including shared documentation, avoiding too much separate QuickBook files containing pieces of documentation and avoiding including too much QuickBook code within the C++ code...

Doxygen aliases

While documenting a large library, it is unavoidable that you have to document the same parameters in different places. For example, the template parameter **Geometry**, and the variable **geometry**, occur at least 100 times in our library.

To avoid repeating the same text again and again, Doxygen aliases are used. So `\tparam_geometry` means that the generic description for a template parameter `geometry` is inserted. `\param_geometry` does the same for a parameter. This is all handled by Doxygen itself. The aliases can also be parameterized, for example: `\return_calc{area}` is expanded to: *The calculated area*

This is for Doxygen alone and is not related to `doxygen_xml2qbk` or QuickBook.

QuickBook macros

QuickBook has the same functionality for the same purpose: macro's or templates can be defined. Within Boost.Geometry this is used in the QuickBook parts of the documentation. So the general rule would be: where it is possible to use a Doxygen alias, we use a Doxygen alias. If we are outside the scope of Doxygen and we want to define a macro, we use a QuickBook macro.

Stated otherwise, we don't use the generated Doxygen documentation, but if we would, it would look correct and would not be unreadable by unexpanded QuickBook macro's.

Code examples

We favour the use of code examples within the generated documentation. Example code must be correct, so examples must compile, run, and produce the correct results. QuickBook has a nice solution to include and present C++ source code, including syntax highlighting. So we generally present the example (a complete example including necessary headerfiles) and the output. Asserts are not used here, these are examples and no tests.

So this is why we did enclose in the \qbk alias above:

```
[heading Example]
[area_with_strategy]
[area_with_strategy_output]
```

We define a heading, we include the example (here denoted by the name "area_with_strategy") and we include the output of the sample "area_with_strategy_output". Note that we simulate that the output is C++ code, a trick giving the appropriate formatting (there might be other ways to get the same effect).

All these QuickBook examples are included in the doc/src/examples/* folders, where also a Jamfile is present. Running bjam there ensures that nothing is broken in the examples.

Some examples, if relevant, are accompanied by images. The images are generated by the example themselves (though marked as commented out for QuickBook), deliver an SVG file which can be manually converted to a PNG (I'm using Inkscape for that which is quite convenient).

Acknowledgments

We like to thank all the people who helped to develop this library.

First of all we are grateful to Hartmut Kaiser for managing the formal review of this library. Hartmut is an excellent review manager, who intervened when necessary and produced the review report quickly.

We thank the 14 reviewers of our library, reviewed from November 5, 2009 to November 22, 2009. Reviews have been written by: Brandon Kohn, Christophe Henry, Fabio Fracassi, Gordon Woodhull, Joachim Faulhaber, Jonathan Franklin, Jose, Lucanus Simonson, Michael Caisse, Michael Fawcett, Paul Bristow, Phil Endecott, Thomas Klimpel, Tom Brinkman.

We also thank all people who discussed on the mailing lists (either at boost, or at osgeo) about Boost.Geometry, in preview stage, or in review stage, or after that.

Finally I (Barend) would like to thank my former employer, Geodan. They allowed me to start a geographic library in 1995, which after a number of incarnations, redesigns, refactorings, previews, a review and even more refactorings have led to the now released Boost.Geometry. And with them I want to thank the team initially involved in developing the library, Valik Solorzano Barboza, Maarten Hilferink, Anne Blankert, and later Sjoerd Schreuder, Steven Fruijtier, Paul den Dulk, and Joris Sierman.