

Generic Programming for Geometry

Barend Gehrels
Geodan Holding B.V.
Amsterdam
the Netherlands

barend.gehrels@geodan.nl

Bruno Lalande
Paris
France

bruno.lalande@gmail.com

Mateusz Loskot
London
UK

mateusz@loskot.net

ABSTRACT

Boost.Geometry has been designed using modern C++ generic programming techniques. The tag dispatching technique is used all over the library, and combined with metafunctions, template metaprogramming, traits, and concept checking. In literature, tag dispatching normally does not get the attention it deserves; this article describes this technique step by step and describes how it can be used and combined with the other techniques, all in the context of geometry, with some trivial geometry functions as examples.

Keywords

Tag dispatching, generic programming, metafunctions, concepts, geometry, Boost

1. INTRODUCTION

Boost.Geometry, a generic library for geometry, is accepted for inclusion into Boost in November '09. This article explains the generic design of Boost.Geometry. It should therefore be interesting for users and developers of Boost.Geometry. Second, maybe even more important, this article serves as an introduction to generic programming using tag dispatching, metafunctions and traits. Third, it explains tag dispatching by type in detail, literature often ignores tag dispatching, or it is described as “dispatch by instance”.

1.1 Geometry

Only some simple geometric shapes will be handled in this article: a point, a segment (a straight line between two points), a linestring (also known as a poly-line, a collection of points or segments) and a multi-linestring (a collection of linestrings, belonging together; representing for example an internally disconnected highway).

There are only trivial algorithms handled: the distance between two points, the length of a segment or a linestring, the number of points of an object.

1.2 Tag Dispatching

Generic programming is explained in different books and articles. Metaprogramming is explained in detail in the book

C++ Template Metaprogramming [2]. However, the technique of tag dispatching is usually not handled in modern C++ books. Both the renowned books C++ Templates [7] and Modern C++ Design [3] do not handle tag dispatching.

Tag Dispatching is described in the web articles Generic Programming in C++ [1], Function Overloading Based on Arbitrary Properties of Types [4]; and in the book C++ Template Metaprogramming [2]. In those descriptions, it is applied as “dispatch by instance”, to select the right overload. The example of `std::advance` is used in all these texts.

In the Boost.Geometry design tag, dispatching is used as “dispatch by type”, to specialize classes on. This article therefore explains tag dispatching in more detail and in combination with other techniques.

2. STEP BY STEP DESIGN

2.1 The Trivial Distance Function

Suppose you need a C++ implementation 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);
}
```

This is simple, sometimes usable, but not generic. For a generic 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 or class), in a Cartesian coordinate system.

A generic library should be able to calculate the distance:

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

BoostCon'10, May 10-14, 2010, Aspen, Colorado, US.

Copyright 2004 ACM 1-58113-000-0/00/0004...\$5.00.

In the subsections below we will make the design step by step more generic.

2.2 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 sqrt(dx * dx + dy * dy);
}
```

This template version is slightly better, but not much. Consider a C++ class where member variables are protected, or have other names. Such a class would not be supported by this function.

2.3 Traits

A generic approach is needed to allow any point type as input to the distance function. So instead of accessing `.x` and `.y` directly, a few levels of indirection are added, using a traits system, introduced in this subsection.

First is described how the distance looks like, when this more generic approach is used:

```
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 sqrt(dx * dx + dy * dy);
}
```

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

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

The referred class `access` should be specialized for our `mypoint` type, implementing a static method called `get`. The primary class template and its specialization for `mypoint` are as following:

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

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

Calling `traits::access<P, 0>::get(a)` now returns the x-coordinate of point `a`. But it is quite verbose for a function like this, used so often in the library. Therefore the function template `get`, listed above, is provided.

So `get<0>(a)` can be called for any point type (having the `traits::access` specialization). It is used as such in the distance algorithm at the start of this subsection.

So we wanted to enable classes with methods like `.x()`, and they are supported, as long as there is a specialization of the access class with a static `get` function returning `.x()` for dimension 0, and `.y()` for dimension 1.

2.4 Dimension Agnosticism

We can now calculate the distance between points in two dimensions, points of any structure or class. However, we wanted to have three dimensions as well. So it has to be made 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.

That can be solved by more usage of templates. The Pythagoras' theorem implementation is moved to a separate class template. This class is specialized for dimension zero to end the recursion. It is defined as following:

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

The distance function is modified to call `pythagoras`, 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 compile-time assertion will prevent point **a** having two dimension and point **b** having three dimensions. The metafunction **dimension**, which is referred to, is defined in another traits class:

```

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

```

Like the access traits structure, this dimension traits structure has to be specialized for **mypoint**. Because it only has to publish a value, it can be conveniently derived from the Boost Metaprogramming Library (MPL, [2]) class `boost::mpl::int_<2>`:

```

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

```

Similar to the function template **get**, the library also contains a shortcut for **dimension**:

```

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

```

This extra definition avoids including the traits namespace each time. But there is a more important reason why it is useful, explained in the section *Different Geometries* below.

Now we have agnosticism in the number of dimensions. Our more generic distance function now accepts points of three or more dimensions.

3. DIFFERENT GEOMETRY TYPES

3.1 Resolving Ambiguity

In the previous section, a dimension agnostic system supporting any point type is designed. But geometry is more than points only.

In this subsection is described how the distance between different geometric types, for example a point and a segment, can be calculated. The formulae are more complex, but the influence on design is even larger.

We do **not** want to add a function with another name:

```

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

```

The library design is generic; the distance function might be called from code not knowing the type of geometry it handles; therefore this function has to be named **distance**.

We neither can create just two overloads, both called **distance**, because that would introduce an ambiguity, two functions having exactly the same template signature.

There are two solutions:

- tag dispatching
- SFINAE (Substitution Failure Is Not An Error)

The first implementation of the library used SFINAE. Later on a switch to tag dispatching was made, among others because SFINAE (to implement concept-based overloading) could not be combined with the Boost Concept Check Library function *requires*. Another big advantage is that tag dispatching can also be used for metafunctions.

3.2 Tag Dispatching

With tag dispatching the distance algorithm inspects the type of geometry of the input parameters.

The distance function is changed into:

```

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 a metafunction called **tag** and forwarding the call to the method **apply** of a `dispatch::distance` class.

The metafunction **tag** is another traits class, which should be specialized for per point type, both shown here:

```

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

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

```

Tags (`point_tag`, `segment_tag`, etc) are empty structures with the sole purpose to specialize a dispatch class, in the design of this library. In other contexts (designs), tags can be instantiated to select the right overload. This is called “dispatch by instance” and more similar to SFINAE.

Here we use “dispatch by type”. The **dispatch** primary class template 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
  >
  class distance {};
}

```

```

template <typename P1, typename P2>
class 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>
class 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 more specializations,
// for point-linestring, etc.

```

The distance algorithm is generic now in the sense that it also supports different geometry types.

One issue: we have to define two dispatch specializations per geometry pair, so different specializations for point - segment and for segment - point. That will also be solved, in the upcoming subsection *reversibility*.

The following example 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);
rgb orange(255, 128, 0);
std::cout << "color distance: "
    << distance(red, orange) << std::endl;

```

3.3 Dispatching Metafunctions

We described above that we had a traits class *dimension*, defined in namespace *traits*, and defined a separate *dimension* class as well. This was 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 metafunction *dimension* for any geometry type, point, segment, polygon, etc. This generic metafunction is different than listed above, and implemented using the same tag dispatching technique:

```

template <typename G>

```

```

struct dimension
    : dispatch::dimension
    <
        typename tag<G>::type, G
    > {};

```

So the call is forwarded to the dispatch function, a bit like with distance, but here it is done using a derived class, while distance had to use an inline function call.

Inside the dispatch namespace this metafunction **dimension** is implemented twice: a generic version and a specialization for point types. The specialization for point types is derived from the traits class. The generic version is derived from the point specialization, as a sort of curiously recurring metafunction. Note that it is not exactly CRTP, because it does not derive from anything, using itself as template parameter. Here the primary template class derives from own its specialization:

```

namespace dispatch {

template <typename Tag, typename G>
struct dimension
    : dimension
    <
        point_tag,
        typename point_type<Tag, G>::type
    > {};

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

```

So with definition, there is a generic metafunction available for all geometry types, implemented using tag dispatching applied on a metafunction.

The metafunction *point_type* is similarly defined for all geometry types, either using the traits system, or using `boost::range_value<G>::type` (for range based geometries).

4. THE POINT CONCEPT FINISHED

4.1 Coordinate Type

Until now we assumed *double*. What if our points are in *integer*, or a high precision type like *tmath* [6]?

The solution is similar. To define the coordinate type, another traits class is added, **coordinate_type**, which should be specialized per point type:

```

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

    template <>
    struct coordinate_type<mypoint>
    {
        typedef double type;
    };

```

Like the generic metafunction **dimension**, a generic version without the traits namespace, valid for any geometry type, is also available for this **coordinate_type** metafunction.

There are two issues. The first issue is that classes like **pythagoras** have two, possibly different, input point types. They might also differ in their coordinate types. Not that that is very likely, but a generic library should handle those strange cases.

One of these coordinate types has to be selected as calculation type, preferably the one with the highest precision. There is a metafunction **select_most_precise** (not worked out here, for brevity) selecting the most precise type of two types.

For **pythagoras** this is sufficient, it does not take the square root, it is just multiplying and adding. It can return a type being the most precise of the coordinate types of its input point types.

The second issue is that if the square root is taken, the distance between two points having *integer* coordinates can be an irrational value (e.g. a *double*). Besides that, a design goal was to avoid square roots. We handle this issue in the subsection *Return Type* later on.

So the modified distance implementation class **pythagoras** using the metafunctions **coordinate_type** and **select_most_precise** becomes:

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

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

4.2 Coordinate System and Strategies

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

Coordinate systems are classes and look like:

```
struct cartesian {};

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

For geographic coordinate systems we can select if its coordinates are stored in degrees or in radians. Other coordinate systems can also be provided with extra information, such as units or a Coordinate Reference System Identifier.

We do not show the implementation of **coordinate_system**, traits, dispatches, it is similar to **coordinate_type** and **dimension**.

The distance function will now change: it will select the computation method for the corresponding coordinate system and then call the dispatch class template 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
        >::type strategy;

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

The used **strategy_distance** is another metafunction, with specializations for different coordinate systems:

```
template <typename T1, typename T2,
    typename P1, typename P2>
struct strategy_distance {};

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

The **pythagoras** class is now declared as a strategy by this metafunction.

For spherical or geographical coordinate systems, another **strategy** (computation method) should be implemented. For spherical coordinate systems we have the haversine formula. So the dispatching traits class template is specialized like this:

```
template <typename P1, typename P2,
    typename U>
struct strategy_distance
    <spherical<U>, spherical<U>,
    P1, P2>
{
    typedef haversine<P1, P2> type;
};
```

For geography, we have some alternatives for distance calculation. There is the Andoyer method (described in

Meeus [5]), fast and precise, and there is the Vincenty method [8], slower and more precise, and there are some less precise approaches as well.

So a strategy is defined per pair of coordinate systems. There are two reasons why this definition is not always enough. First, library users might wish to use another strategy. Second, the strategy declaration within the function template **distance** does not have construction parameters. Construction parameters are essential for strategies implementing the Andoyer, Vincenty or haversine methods, having a constructor taking the radius of the earth as a parameter

Therefore, an overload is added for the distance function template, with a strategy object as a third parameter:

```
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 itself has to have a method taking two points as arguments (for points). This method also here is called **apply**. It is not required that it is a static method, because of the constructor and possible member variables..

We can call distance like this:

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

where c1,c2 are Cartesian points.

Or like this:

```
distance(track1, track2)
```

where t1 and t2 are linestrings (GPS tracks) defined in the geographic coordinate system, using the default strategy for Geographic shapes (defined above as being Andoyer).

Finally it can be called like this:

```
distance(t1, t2, vincenty<G1, G2>(6275))
```

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

4.3 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*.

5. MORE SPECIFIC SOLUTIONS

This section solves some issues mentioned in the previous sections. They are specific to some algorithms, such as the **distance** algorithm, so they do not need to be present in any design similar to the one described in this article.

First the return type is introduced. Then the subsection *reversibility* shows how functions with two possibly different geometry arguments can be implemented without making duplicated dispatches of all combinations.

5.1 Return Type

The return type of distance was problematic before because of:

- coordinate types: the result of points stored in *integer* might be a *double*
- the square root: taking it can sometimes be avoided

Square root is a relatively expensive function and should be avoided if not necessary, though this argument was more important in the past than it is now. It is not necessary when comparing distances, and comparing is ubiquitous.

Therefore, a class is defined that contains the squared value and is convertible to (for example) a *double* value. When converted, the square root is taken (with a trivial enhancement: only if not done before).

This, however, only has to be done for the Pythagoras' theorem. Spherical distance functions do not take the square root so they can just return their calculated distance.

The distance result class is hence dependant on strategy and made a member type of the strategy. The version for the Cartesian coordinate system is looking like:

```
template<typename T = double>
class 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, e.g.:

```
typedef cartesian_distance<C> return_type
or:
typedef C return_type
```

for Pythagoras and Andoyer (spherical), respectively, where C is the coordinate type of the two geometry types, defined with **select_most_precise**, and after that promoted to a floating point type.

Note that a strategy is not a metafunction, there is no member type defined with the name **type**.

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 overload without strategy we have to select strategy, coordinate types, and select the most precise of them. It would be spacious to do it in one line. For this reason we create another metafunction:

```
template <typename G1, typename G2>
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 use it in the distance function:

```
template <typename G1, typename G2>
inline typename distance_result
<G1, G2>::type
distance(G1 const& G1, G2 const& G2)
```

5.2 Reversibility

Our `dispatch::distance` class was specialized for pairs of geometries, like `<point_tag, point_tag>` or `<point_tag, segment_tag>`. But library users can also call the distance function with a segment and a point, in that order. Actually, there are many geometry types (polygon, box, linestring), how to implement all combinations without duplicating all tag dispatching classes?

The answer is to automatically reverse arguments, if appropriate. For distance it is appropriate because distance is a commutative function.

We add a metafunction **geometry_id**, which has specializations for each geometry type. It should define a number such that it can be ordered. Therefore it is derived from `boost::mpl::int_`. The numbers themselves are not important, but it is important that all specializations have their template parameters ordered conform the defined numbering.

Then we add a metafunction **reverse_dispatch**:

```
template <typename G1, typename G2>
struct reverse_dispatch
: boost::mpl::if_c
<
  geometry_id<G1>::value >
  geometry_id<G2>::value
  true_type,
  false_type
> {};
```

To use this, the function **distance** will be modified again, using some template metaprogramming:

```
return boost::mpl::if_
<
  typename
    reverse_dispatch<G1, G2>::type,
  detail::distance_reversed
  <
    typename tag<G1>::type,
    typename tag<G2>::type,
    G1, G2,
    Strategy
  >,
  dispatch::distance
  <
    typename tag<G1>::type,
    typename tag<G2>::type,
    G1, G2,
    Strategy
  >
>::type::apply(g1, g2, s);
```

In this definition, the `detail::distance_reversed` class calls `dispatch::distance`, with all its template arguments and function arguments reversed.

6. POLICY AND REUSE

This section shows how one implementation can be used as a policy in another implementation. This is showed implementing the design created before, using all the described techniques, for the length function.

6.1 Tag Dispatching

We implement a function **length** returning the length of any geometry, for example a line segment (of two points), a linestring, or a multi-linestring. We start with the generic function template **length**:

```

template<typename G>
inline typename length_result<G>::type
length(G const& geometry)
{
    // Get strategy for distance: S (...)
    return dispatch::length
        <
            typename tag<G>::type, G, S
        >::apply(geometry);
}

```

This is similar to how we ended with the distance function, but the function **length** is simpler than the function **distance** because it has one parameter, and thus one template parameter.

The function template **length** dispatches its call to the class specialized for the tag which corresponds to the geometry type. The primary class template and the partial specialization for **linestring** are shown here:

```

namespace dispatch {

template <typename T, typename G,
          typename S>
class length
: detail::calculate_null
<
    typename length_result<G>::type,
    G, S
> {};

template <typename G, typename S>
class length<linestring_tag, G, S>
: detail::range_length<G, S> {};
}

```

The specialization for **linestring** does not contain the actual implementation, but it forwards (by derivation) to another class, in namespace **detail**. All this is zero cost, there are only derivations, it is all compile-time.

6.2 Implementation Detail

Namespace **detail** contains the implementation of the calculation of the length of any range. The implementation can be used for a **linestring**, but also for other ranges (e.g. to calculate the perimeter of a ring of points).

The (partial) implementation is shown here:

```

template<typename R, typename S>
class range_length
{
    typedef typename length_result<R>::type
        return_type;

    static return_type apply(R const& range,
                             S const& strategy)
    {
        return_type sum = return_type();
        // implementation: walk through
        // range and sum distances between
        // segments, using the strategy
        return sum;
    }
};

```

6.3 Multi Linestring

A so-called **multi_linestring** is a collection of **linestrings**. U.S. Highway 2, for example, is defined as one highway, but consists of two separate sections [9]. In between of those two, there is Canada.

This subsection describes how to calculate its length using a generic implementation.

We could create a specialization of **dispatch::length** for the **multi_linestring_tag**, walk over the elements of the **multi_linestring** collection there and sum the result. But if we do that, and later on implement perimeter or area, and the multi-versions of those, we would end with several similar implementations.

So we take a different approach and create a **multi_sum** implementation:

```

template <typename RT, typename MG,
          typename S, typename Policy>
class multi_sum
{
    static RT apply(MG const& multi,
                   S const& strat)
    {
        RT sum = RT();
        for (typename
            range_iterator<const MG>::type
                it = begin(multi);
                it != end(multi); ++it)
        {
            sum += Policy::apply(*it, strat);
        }
        return sum;
    }
};

```

Where **RT** is return type, **MG** is the multi-geometry type, **S** is the strategy (for distance). It calculates the sum for any implementation, defined as a policy.

We finally show the specialization of length calculation for a **multi_linestring**:

```

template <typename ML, typename S>
class length<multi_linestring_tag, ML, S>
: multi_sum
<
    typename length_result<ML>::type,
    ML, S,
    range_length
    <
        typename range_value<ML>::type,
        S
    >
> {};

```

The length dispatch specialization is empty; it just derives from the defined **multi-sum**. For area, perimeter, for all algorithms summing values, of all multi-geometries, we can do the same.

6.4 Apply

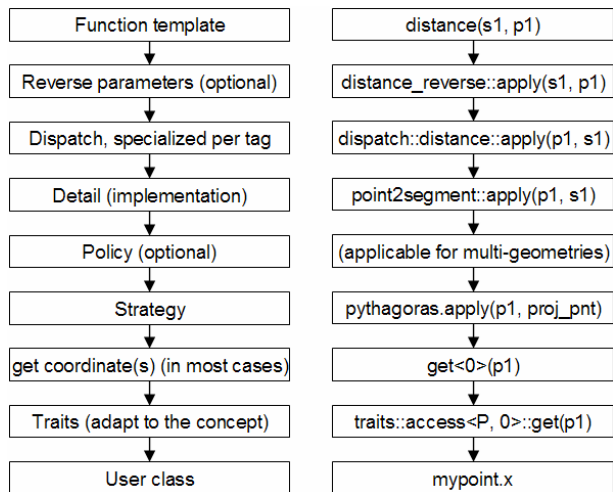
Of course, when using one class as a policy in other classes, all those classes should have the same signature. They should have a method **apply** with the same number of parameters.

The Boost.Geometry library follows this convention. It is essential that all dispatch structures, implementation details, policies and strategies have a method **apply**.

Method **apply** is not implemented as operator() because it is a static method in most cases.

7. THE CHAIN COMPLETE

It is useful to schematize how a call would forward through the design. All calls are implemented via derivation (zero cost) or via inline function calls (zero cost, if the compiler does his job). The scheme below follows a call of function distance (with a segment and a point), via dispatch to the implementation details; and from there, to get coordinates, via the function template **get** to the class containing the point, calling its .x member variable.



In this scheme, the proj_pnt stands for the projected point, an intermediary object used in the calculation of the distance point-segment.

8. METAFUNCTION FINE-TUNING

This section describes two useful additional techniques when designing a system using tag dispatching.

8.1 Fine-tuning by similarity

Suppose you want to implement a simple algorithm returning the number of points of a geometry (num_points).

This can be implemented the same way, using tag dispatching, a point returns 1, a segment 2, a linestring the number of elements in the collection, a polygon (without holes) the number of elements in the collection... Yes, the last two would have the same implementation. For the number of points, the num_points dispatch specialization would be the same.

We can avoid this using the fact that a linestring and a polygon without holes are similar in some aspects. So we implement this similarity in the metafunction, for example called **is_range**.

The num_point dispatch structure has an extra template parameter, **IsRange**, and function template **num_points** provides this extra parameter specifying the metafunction **is_range**.

We call this **metafunction fine-tuning**. Many similarities can be handled this way, the next sub-section explains how it can be applied for multi-geometries.

8.2 Dispatch as policy

So if we would specialize the length dispatch class for another multi-geometry, for example a multi-segment, the lengths would be summed again, it would be exactly the same derivation. Though similar derivations are less problematic than similar implementations, it is nevertheless useful if it can be avoided.

Therefore, the metafunction fine-tuning technique just described can be used. The dispatch specializations are shared using an additional template parameter **IsMulti**, provided in the function template **length**, using the metafunction **is_multi**.

So the primary class template of length dispatch would be:

```

template <typename T, typename G,
         typename S, bool IsMulti>
class length
: detail::calculate_null
<
    typename length_result<G>::type,
    G, S
> {};
  
```

and the specialization for multi-geometries would be:

```

template <typename MT, typename MG,
         typename S>
class length<MT, MG, S, true>
: multi_sum
<
    typename length_result<MG>::type,
    MG, S,
    length
    <
        typename tag
        <
            typename
            range_value<MG>::type
            >::type,
            typename range_value<MG>::type,
            S, false
        >
    >
> {};
  
```

So in this case the dispatch structure specialized for the single version is used as a policy for the dispatch structure specialized for any multi-geometry. This is the second curiously recurring pattern we encounter, and here it is indeed a form of CRTP, because dispatch::length is derived from multi_sum, specifying dispatch::length as a template argument.

9. CONCEPT CHECKING

Section 4 above handled the point concept, enabling users to provide their own geometries which can be adapted to concepts via traits classes. This is available for points but also for linestrings and other geometries.

Handling the geometry concepts completely is out of the scope of this article. But it is useful to describe how the concepts can be checked.

Boost.Geometry uses the Boost Concept Check Library (BCCL) to check the concepts. Checking concepts should be done as early as possible, to give the most meaningful error messages in the case that the provided classes do not fulfill the concepts.

The first entry point is the call to the function (e.g. distance). But the library cannot call BOOST_CONCEPT_ASSERT there, because it is not yet known if the parameter contains a point type (Point Concept) or a segment type (Segment Concept) or another geometry type.

It is possible to check the concept in the dispatch specializations. But this would result in similar lines in all specializations, everywhere, and besides that it is not possible where dispatch classes are shared using metafunction fine-tuning.

Therefore we add an additional class template, specialized again on geometry tags. This class template does the concept checking. So concept checking is only done once within the distance algorithm, in the function template, calling:

```
concept::check<Geometry1 const>();
concept::check<Geometry2 const>();
```

This is forwarding (via a function template, adding a compile-time boolean template parameter to distinguish points and const points, Point Concepts and Const Point Concepts) to:

```
template <typename Geometry, bool IsConst>
struct checker : dispatch::check
{
    <
        typename tag<Geometry>::type,
        Geometry,
        IsConst
    > {};
```

The specialization for point then checks as the following:

```
template <typename P>
struct check<point_tag, P, true>
: detail::concept_check::check
<concept::ConstPoint<P> > {};
```

Where the detail concept checker referred to is checking the concept:

```
template <typename Concept>
class check
{
    BOOST_CONCEPT_ASSERT((Concept ));
};
```

10. CONCLUSIONS

Boost.Geometry is designed using tag dispatching, concepts, traits, metaprogramming and all the techniques described step by step in this article. All these techniques have been proven indispensable. Algorithm implementations, metafunctions and traits are distinguished using tag dispatching. Dispatch classes are shared (fine-tuned) with help of metafunctions, and by automatically reversing template arguments. Implementation classes or dispatch classes are reused as policies in other dispatch classes, sometimes in the form of CRTP.

Modern C++ literature often ignores tag dispatching, but it deserves more attention because it is a wonderful technique for designing complex systems, and this article hopes to help here by drawing some attention.

11. REFERENCES

- [1] Abrahams, D. and Gregor, D. 2001 Generic Programming in C++: Techniques. <http://www.generic-programming.org/languages/cpp/techniques.php>
- [2] Abrahams, D. and Gurtovoy, A. 2004. C++ Template Metaprogramming. Addison-Wesley.
- [3] Alexandrescu, A. 2001. Modern C++ Design. Addison-Wesley.
- [4] Järvi, J. Willcock, J., Hinnant, H., Lumsdaine, A. 2003. Function Overloading Based on Arbitrary Properties of Types. C/C++ Users Journal, June 01, 2003
- [5] Meeus, J. 1991. Astronomical Algorithms. Willmann-Bell
- [6] TTMath, Bignum C++ library, <http://www.ttmath.org>
- [7] Vandevoorde, D. and Josuttis, N.M. 2003. C++ Templates: The Complete Guide. Addison-Wesley.
- [8] Vincenty, T. 1975. Direct and inverse solutions of geodesics on the ellipsoid with application of nested equations Survey Review 22 (176): 88-93. http://www.ngs.noaa.gov/PUBS_LIB/inverse.pdf
- [9] Wikipedia, U.S. Route 2, http://en.wikipedia.org/wiki/U.S._Route_2