

# Boost.Geometryに学ぶ テンプレートライブラリの設計

高橋 晶(Akira Takahashi)

[id:faith\\_and\\_brave](#)

[@cpp\\_akira](#)

Boost.勉強会 #6 札幌 2011/11/05(土)

- 株式会社ロングゲート 取締役
- C++標準化委員会エキスパートメンバ
- Boost Geometry Libraryコントリビュータ
- boostjpコミュニティ マネージャ
- 著書『C++テンプレートテクニック』
- 『プログラミングの魔導書』編集長

- 今回の発表では、
  - Boost.Geometryがどのような使い方ができ、
  - どのような設計になっているのかを知り、
  - こういったライブラリをどうすれば作れるのかを見ていきます。

1. Boost Geometry Libraryとは
2. ジェネリックプログラミング
3. Concept-based Design

## Chapter 01

# Boost Geometry Libraryとは

*What's Boost.Geometry?*

- Boost Geometry Libraryは、計算幾何のためのライブラリ
- 作者:Barend Gehrels
- Boost 1.47.0でリリースされた。

Boost.Geometryのアルゴリズムは、  
複数のモデルに対して動作する。

```
const linestring a = assign::list_of<point>(0, 2)(2, 2);  
const linestring b = assign::list_of<point>(1, 0)(1, 4);
```

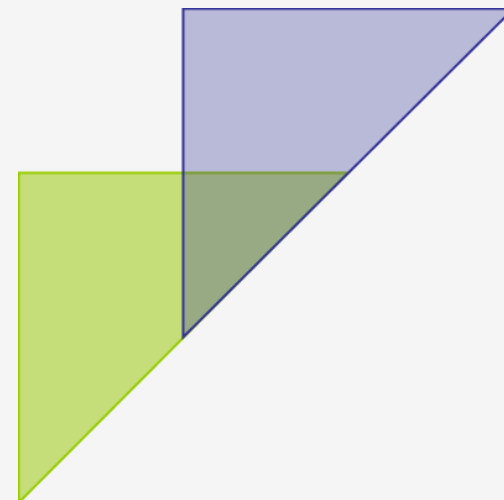
// 2つの線が交わっているか

```
const bool result = geometry::intersects(a, b);  
BOOST_ASSERT(result);
```



Boost.Geometryのアルゴリズムは、  
複数のモデルに対して動作する。

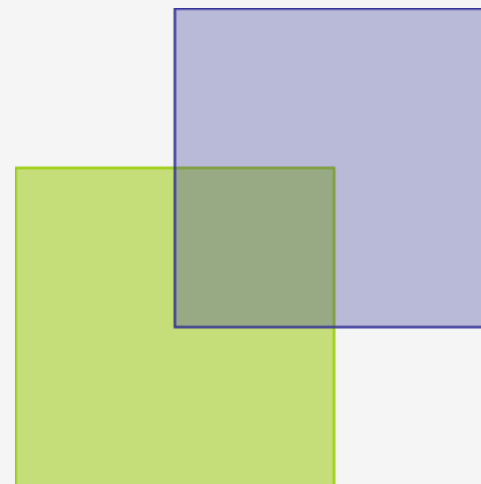
```
polygon a, b;  
geometry::exterior_ring(a) =  
    assign::list_of<point>(0, 0)(3, 3)(0, 3)(0, 0);  
  
geometry::exterior_ring(b) =  
    assign::list_of<point>(1.5, 1.5)(4.5, 4.5)(1.5, 4.5)(1.5, 1.5);  
  
// 2つのポリゴンが交わっているか  
const bool result = geometry::intersects(a, b);  
BOOST_ASSERT(result);
```





Boost.Geometryのアルゴリズムは、  
複数のモデルに対して動作する。

```
const box a(point(0, 0),    point(3, 3));  
const box b(point(1.5, 1.5), point(4.5, 4.5));  
  
// 2つの四角形が交わっているか  
const bool result = geometry::intersects(a, b);  
BOOST_ASSERT(result);
```



Boost.Geometryの各モデルは、ユーザー定義型に適用できるため、Boost.Geometryのアルゴリズムを他のライブラリの型に対して使用することができる。

```
struct Point {  
    int x, y;  
  
    Point() : x(0), y(0) {}  
    Point(int x, int y) : x(x), y(y) {}  
};
```

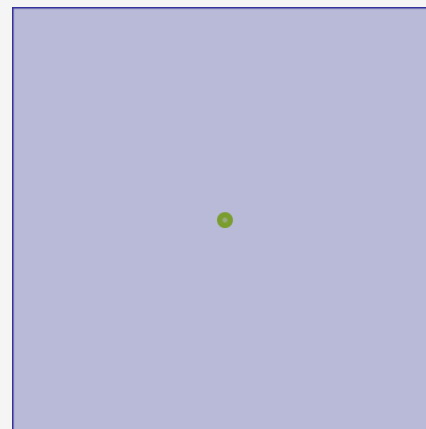
```
struct Rect {  
    int left, top, right, bottom;  
  
    Rect() : left(0), top(0), right(0), bottom(0) {}  
    Rect(int left, int top, int right, int bottom)  
        : left(left), top(top), right(right), bottom(bottom) {}  
};
```

Boost.Geometryの各モデルは、ユーザー定義型に適用できるため、Boost.Geometryのアルゴリズムを他のライブラリの型に対して使用することができる。

```
BOOST_GEOMETRY_REGISTER_POINT_2D(Point, int, cs::cartesian, x, y);  
BOOST_GEOMETRY_REGISTER_BOX_2D_4VALUES(Rect, Point, left, top, right,  
bottom);
```

```
const Point p(1, 1);  
const Rect box(0, 0, 2, 2);
```

```
// 点が四角形の内側にあるか  
const bool result = geometry::within(p, box);  
BOOST_ASSERT(result);
```



- Boost.Geometryの設計は、
  1. あらゆるモデルに対して同じアルゴリズムを適用でき、
  2. 他のライブラリのモデルに対してBoost.Geometryのアルゴリズムを適用できる。
- この発表では、このような設計のライブラリをどうやって作るのかを見ていく。

Chapter 02

# ジェネリックプログラミング

*Generic Programming*

- データ型に依存しないプログラミング手法。
- C++では、テンプレートという、パラメタライズドタイプの機能によって実現する。
- オブジェクト指向的なライブラリは、継承関係にあるデータを扱うが、テンプレートによって設計されたライブラリは、継承関係に依存せずにあらゆるデータを扱う設計が可能となる。

以下のmin関数は、operator<が使用できる  
あらゆる型で振る舞う関数

```
template <class T>
T min(T a, T b)
{
    return a < b ? a : b;
}
```

```
int    x = min(1, 2);      // Tはintに置き換えられる
double d = min(1.0, 2.0); // Tはdoubleに置き換えられる
char   c = min('1', '1'); // Tはcharに置き換えられる
```

以下のListクラスは、あらゆるデータ型を保持できるコンテナ

```
template <class T>
class List {
    T* data_;
    size_t size_;
public:
    ...
    void add(const T& x)
    {
        T* tmp = new T[size_ + 1];
        for (size_t i = 0; i < size_; ++i) { tmp[i] = data_[i]; }
        delete[] data_;

        data_ = tmp;
        data_[size_++] = x;
    }

    T& operator[](size_t i) { return data_[i]; }
    size_t size() const { return size_; }
};
```



以下のListクラスは、あらゆるデータ型を保持できるコンテナ

```
List<int> ls;  
ls.add(3);  
ls.add(1);  
ls.add(4);  
  
for (size_t i = 0; i < ls.size(); ++i) {  
    cout << ls[i] << endl;  
}
```

```
List<string> ls;  
ls.add("abc");  
ls.add("hello");  
ls.add("goodbye");  
  
for (size_t i = 0; i < ls.size(); ++i) {  
    cout << ls[i] << endl;  
}
```

- Alexander Stepanovが設計したSTLは、テンプレートによって、データ構造とアルゴリズムの分離を行った。
- これによって、アルゴリズムはあらゆるデータ構造に対して振る舞うことが可能になったため、アルゴリズムは誰か一人が書けばよくなった。

- データ型が異なる以外は同じコードになるのであれば、それはジェネリック(汎用的)にできる。
- 型によって最適化を行いたい場合は特殊化することができる。
- ジェネリックプログラミングによるコードの汎用化は、ユーザーコード量を限りなく減らすことができる。

Chapter 03

# コンセプトに基づく設計

*Concept-based Design*

- Boost.Geometryには、Point, LineString, Polygon, Boxなどの「**コンセプト**」と呼ばれる分類が存在する。
- これらコンセプトは、class point; のような具体的な型ではなく、「Pointと見なせるあらゆる型」を意味する。
- Boost.GeometryはPointコンセプトを満たす型として `boost::geometry::model::d2::point_xy<T>` という2次元の点を表す型を提供している。
- Boost.Geometryでは、point\_xyという型だけでなく、その他のユーザーが定義した型を、Pointコンセプトの型としても扱うことができる。

これらのPointコンセプトを満たす型は継承関係にある必要はないし、インタフェースも異なるが、同じように扱うことができる。

```
polygon<point_xy<double>> poly;  
double result = area(poly);
```

```
polygon<std::pair<double, double>> poly;  
double result = area(poly);
```

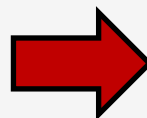
```
polygon<UserDefinedPointType> poly;  
double result = area(poly);
```

これらのPointコンセプトを満たす型は継承関係にある必要はないし、インタフェースも異なるが、同じように扱うことができる。

```
polygon<point_xy<double>> poly;  
double result = area(poly);
```

```
polygon<std::pair<double, double>> poly;  
double result = area(poly);
```

```
polygon<UserDefinedPointType> poly;  
double result = area(poly);
```



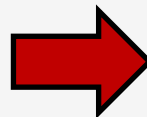
```
template <class T>  
struct point_xy {  
    T values[2];  
};
```

これらのPointコンセプトを満たす型は継承関係にある必要はないし、インタフェースも異なるが、同じように扱うことができる。

```
polygon<point_xy<double>> poly;  
double result = area(poly);
```

```
polygon<std::pair<double, double>> poly;  
double result = area(poly);
```

```
polygon<UserDefinedPointType> poly;  
double result = area(poly);
```



```
template <class T1, class T2>  
struct pair {  
    T1 first;  
    T2 second;  
};
```

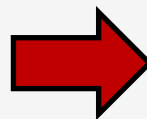


これらのPointコンセプトを満たす型は継承関係にある必要はないし、インタフェースも異なるが、同じように扱うことができる。

```
polygon<point_xy<double>> poly;  
double result = area(poly);
```

```
polygon<std::pair<double, double>> poly;  
double result = area(poly);
```

```
polygon<UserDefinedPointType> poly;  
double result = area(poly);
```



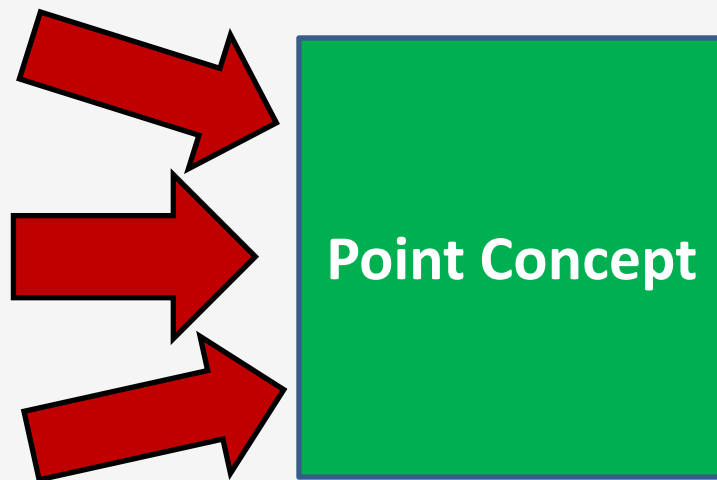
```
struct UserDefinedPointType {  
    ????  
};
```

これらのPointコンセプトを満たす型は継承関係にある必要はないし、インタフェースも異なるが、同じように扱うことができる。

```
polygon<point_xy<double>> poly;  
double result = area(poly);
```

```
polygon<std::pair<double, double>> poly;  
double result = area(poly);
```

```
polygon<UserDefinedPointType> poly;  
double result = area(poly);
```



こういった設計はどうやったら実現できるのかを紹介する

通常、点を表すクラスは以下のように定義することになる:

```
template <class T>
struct point {
    T x, y;
};
```

コンセプトベースなライブラリの場合は、以下のようにアダプト可能なインタフェースとして定義する:

```
template <class T>
struct point_traits {
    typedef typename T::value_type value_type;

    static value_type get_x(const T& p) { return p.x; }
    static value_type get_y(const T& p) { return p.y; }

    static T construct(value_type x_, value_type y_)
        { return T(x_, y_); }

    static T subtract(const T& a, const T& b)
        { return construct(a.x - b.x, a.y - b.y); }
};
```

Pointを操作するアルゴリズムは、点のクラスを直接操作せず、traitsを通して行う。

```
template <class Point>
double distance(const Point& a, const Point& b)
{
    const Point d = a - b;
    return std::sqrt(d.x * d.x + d.y * d.y);
}
```



```
template <class Point>
double distance(const Point& a, const Point& b)
{
    typedef point_traits<Point> traits;
    const Point d = traits::subtract(a, b);
    return std::sqrt(
        traits::get_x(d) * traits::get_x(d) +
        traits::get_y(d) * traits::get_y(d));
}
```

ユーザー定義の点を表す型をPointコンセプトを満たすようにアダプトする:

```
struct my_point {  
    double data[2];  
};  
  
template <>  
struct point_traits<my_point> {  
    typedef double value_type;  
  
    static value_type get_x(const my_point& p) { return p.data[0]; }  
    static value_type get_y(const my_point& p) { return p.data[1]; }  
  
    static my_point construct(double x, double y)  
        { my_point p = {x, y}; return p; }  
  
    static my_point subtract(const my_point& a, const my_point& b)  
        { return constrcut(a[0] - b[0], a[1] - b[1]); }  
};
```

これで、distanceアルゴリズムは、point\_traitsでアダプトしたあらゆる型で振る舞うことができるようになる。

```
my_point p = { 0.0, 0.0 };  
my_point q = { 3.0, 3.0 };  
  
double d = distance(p, q); // 4.24
```

コンセプトベースライブラリを作る基本的な流れ:

1. 具体的な型を直接操作しない
2. アダプト可能なインタフェースを定義する
3. アルゴリズムでは、traitsを通して間接的に操作を行う
4. ユーザー定義型をtraitsにアダプトする

Boost.Geometryは複数のコンセプトをサポートしているので、アルゴリズムはコンセプトごとに最適な実装を選択させる必要がある。

実現したいこと:

```
template <class Point, class Point>
double distance(Point, Point);

template <class Point, class LineString>
double distance(Point, LineString);

template <class LineString, class Point>
double distance(LineString, Point);
```

これを実現するには、「**タグディスパッチ**」という手法を用いる。



まず、PointコンセプトとLineStringコンセプト、それぞれのためのタグを定義する。

タグは単なる空のクラス。

```
struct point_tag {};  
struct line_string_tag {};
```

タグを返すメタ関数を用意する。  
中身なし。

```
template <class T>  
class get_tag;
```

ユーザー定義の型をアダプトする際に、get\_tagを特殊化する。

```
template <>
struct get_tag<my_point> {
    typedef point_tag type;
};

template <>
struct get_tag<my_line> {
    typedef line_string_tag type;
};
```

アルゴリズムはあらゆる型を受け取れるようにし、  
型のタグを取得して専門特化した関数に転送する。

```
template <class Geometry1, class Geometry2>
double distance(const Geometry1& a, const Geometry2& b)
{
    return distance(a, b,
                    typename get_tag<Geometry1>::type(),
                    typename get_tag<Geometry2>::type());
}
```

## 組み合わせごとのアルゴリズムを定義する

```
// 点と点
template <class Point>
double distance(const Point& a, const Point& b,
               point_tag, point_tag)
{
    typedef point_traits<Point> traits;
    const Point d = traits::subtract(a, b);
    return std::sqrt(
        traits::get_x(d) * traits1::get_x(d) +
        traits::get_y(d) * traits2::get_y(d));
}
```

## 組み合わせごとのアルゴリズムを定義する

```
// 点と線
template <class Point, class LineString>
double distance(const Point& a, const LineString& b,
               point_tag, line_string_tag)
{
    typedef line_string_traits<LineString> traits;
    return std::min(
        distance(a, traits::get_start(b)),
        distance(a, traits::get_last(b))
    );
}

// 線と点
template <class LineString, class Point>
double distance(const LineString& a, const Point& b,
               line_string_tag, point_tag)
{ return distance(b, a); }
```

これで、PointとLineStringの組み合わせごとに最適な実装を切り替えることができるようになった。

```
// 点と点  
my_point p = {0.0, 0.0};  
my_point q = {3.0, 3.0};  
  
double d = distance(p, q);
```

```
// 点と線  
my_point p = {0.0, 0.0};  
my_line l = {{2.0, 2.0}, {3.0, 3.0}};  
  
double d = distance(p, l); // 逆もできる : distance(l, p);
```

コンセプトでオーバーロードする流れ:

1. コンセプトを表すタグを定義する。タグは単なる空のクラス
2. 型を受け取ってタグを返すメタ関数を定義する
3. タグを返すメタ関数を、型ごとに特殊化する
4. 関数テンプレートのテンプレートパラメータを、タグを返すメタ関数に渡し、結果の型を転送する
5. タグをパラメータにとるオーバーロードを用意する



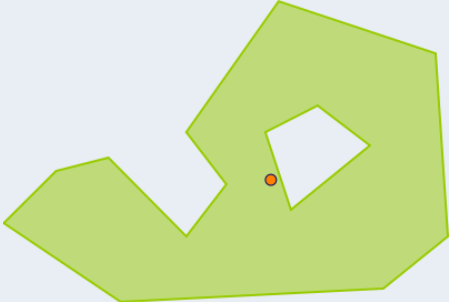
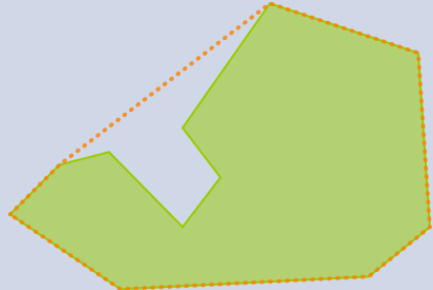
- Boost.Graph  
抽象化が難しいと言われるグラフにおいて、データ構造とアルゴリズムをSTLの概念に基づいて分離したライブラリ。  
特性別にグラフ構造を分類するのにコンセプトを採用している。
- Boost.Fusion  
タプルを異なる型のシーケンスと見なしイテレート可能にしたライブラリ。  
Fusion Sequenceというコンセプトにアダプトされたあらゆる型を、Fusionのタプルと見なして数々のアルゴリズムを適用することができる。

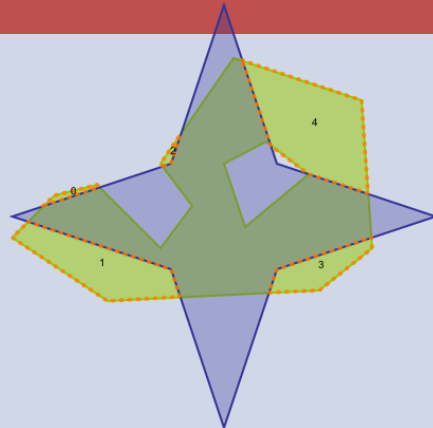
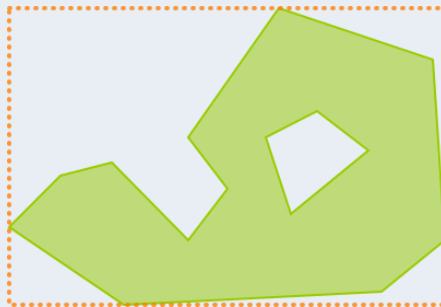
- Boost.Geometryは、有用なアルゴリズムを提供するだけではなく、既存の他のライブラリと組み合わせて使用することを想定して設計されている。  
そういった設計には、コンセプトという考えを導入すると拡張性が高くなる。
- データとアルゴリズムを分離したことで誰か一人がアルゴリズムを書けばいいなら、みんなで知恵を絞ることで、そのライブラリを使用している全てのソフトウェアが改善されることを期待できる。
- 今回の話をふまえて、ぜひテンプレートライブラリを作ってみてください！

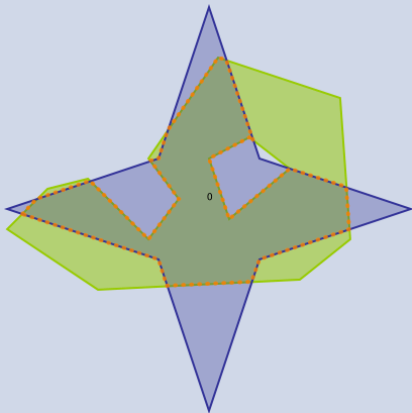
Appendix

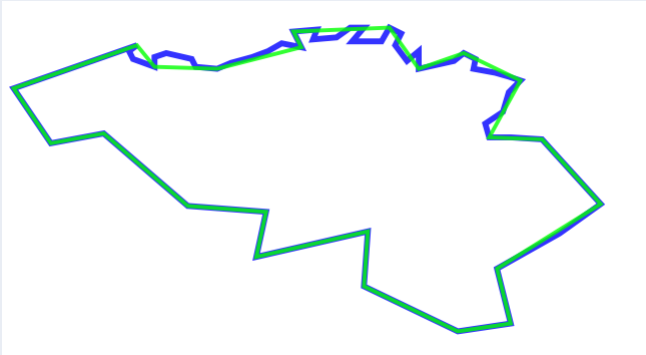
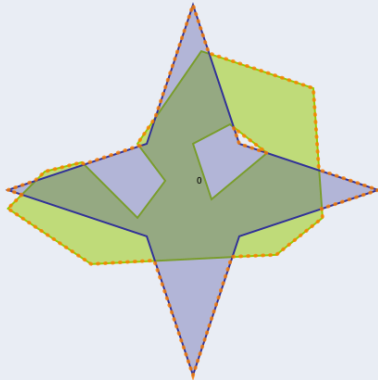
# アルゴリズムとモデル一覧

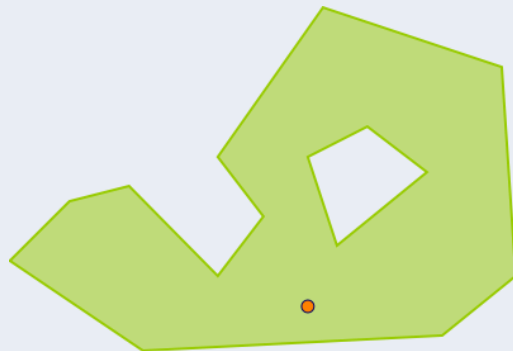
*Algorithm and Model List*

関数	説明	図
area	面積を求める	
centroid	図形の中心点を求める	
convex_hull	凸包を求める	
correct	図形の向きを修正し、終了点を補完する	

関数	説明	図
difference	2つの図形の差異を求める	
disjoint	2つの図形が互いに素か判定する	
distance	2つの図形の距離を求める	
envelope	包絡線を求める	
equals	2つの図形が等しいか判定する	

関数	説明	図
expand	他の図形でboxを拡張する	
for_each_point for_each_segment	図形を走査する	
intersection	2つの図形の共通部分を求める	
intersects	2つの図形が交わっているか判定する	
length perimeter	図形の長さを求める	
overlaps	2つの図形が重なっているかを判定する	

関数	説明	図
reverse	図形を逆向きにする	
simplify	図形を簡略化する	
transform	図形の単位変換などを行う	
union_	2つの図形の和を求める	

関数	説明	図
unique	図形内の重複を削除する	
within	図形がもう一方の図形の 内側にあるか判定する	



モデル	説明
Point	N次元の点
LineString	複数の線分
Polygon	三角形(メッシュ構造)
Box	四角形
Ring	輪
Segment	線分
MultiPoint	複数の点
MultiLineString	複数の線
MultiPolygon	複数の三角形