

Boost Fusion Library

高橋 晶(Takahashi Akira)

[id:faith_and_brave](#)

[@cpp_akira](#)

Boost.勉強会 #4 2011/02/26(土)

- Boost.Fusionは、ドキュメントはしっかり書かれている。
- しかし、そのドキュメントだけ見ても何に使えばいいのかはさっぱりわからない。
- この発表では、Boost.Fusionをひと通り見て回り、その後このライブラリをどんなケースで使用するのかを解説していきます。

- Boost.Fusionとは
- Fusionシーケンス
- 無名ユーザ一定義型と名前ありタプル
- Fusionの使いどころ

Chapter.01

***Boost.Fusion*とは何か**

- タプルのデータ構造とアルゴリズムのライブラリ。
- Python, Scheme, Haskellなどにあるヘテロなコンテナを表現するために作られた。
- Boost.Fusionは元々、Boost.Spiritに含まれていた。

- Boost Tuple Libraryなどですでに実装されている、`std::pair`(組)のN個バージョン。
- 組は2要素のみ格納できるが、タプルはN要素格納できる。

例:

```
pair<int, char> p(1, 'a');
```

```
tuple<int, char, string> t(1, 'a', "Hello");
```

ヘテロなコンテナとは

あらゆる型を格納できるコンテナ。

`std::vector<boost::any>`などでも表現できるが、
タプルもヘテロなコンテナと見なすことができる。

```
tuple<int> t1(1);
```

```
tuple<int, char> t2(get<0>(t1), 'a'); // 要素を追加
```


- タプルをヘテロなコンテナと見なすことで、タプルに対して、transform(map), accumulate(fold)といった有用なアルゴリズムを適用するアイデアが出てくる。
- Boost.Fusionは、STLの概念(コンテナ、イテレータ、アルゴリズム)に基づいて、タプルに対する多くの有用なアルゴリズムを提供する。


```
#include <iostream>
#include <boost/fusion/include/vector.hpp>
#include <boost/fusion/include/for_each.hpp>

namespace fusion = boost::fusion;

struct disper {
    template <class T>
    void operator()(const T& x) const
    {
        std::cout << x << std::endl;
    }
};

int main()
{
    fusion::vector<int, char, double> v(1, 'a', 3.14);
    fusion::for_each(v, disper());
}
```



```
1
a
3.14
```

Chapter.02

***Fusion*シーケンス**

Boost.Fusionでは、タプルを
「異なる型を格納するリスト」と見なす。

リストは、同じ型の異なる値を格納する。

型 : `std::vector<int>`

値 : {1, 2, 3...}

タプル: 異なる型の値を格納する。

型 : `boost::fusion::vector<int, char, std::string>`

値 : (1, 'a', "Hello"...)

種類	型	補足
Random Access Sequence	vector	要素にランダムアクセス可能なシーケンス。デフォルトで使用するべき型。
Forward Sequence	list	前方向に走査可能なシーケンス。
Bidirectional Sequence	deque	双方向に走査可能なシーケンス。 ただしアンドキュメント。

実際はvectorしか使わないと考えていい。

list(というかcons)はいちおうBoost.Spirit.(Qi | Karma)で使われている。

dequeを使っている人は見たことがない。

ランダムアクセスな型リスト(タプル)の実装方法は、『C++テンプレートメタプログラミング』を参照。添字で特殊化している。

テンプレートの再帰が必要なくなるので、

Forward SequenceよりRandom Access Sequenceの方がコンパイルが速い。

for_each等でループのアンロールもしやすい。

FusionシーケンスはSTLと同様、イテレータのインタフェースを持つ。
イテレータは進むたびに異なる型を指す。

```
typedef fusion::vector<int, char, double> vec;
```

```
const vec v(1, 'a', 3.14);
```

```
fusion::vector_iterator<const vec, 0> first  = fusion::begin(v);  
fusion::vector_iterator<const vec, 3> last   = fusion::end(v);  
fusion::vector_iterator<const vec, 1> second = fusion::next(first);
```

```
BOOST_ASSERT(fusion::deref(first) == 1);  
BOOST_ASSERT(fusion::deref(second) == 'a');
```

これで、シーケンスとアルゴリズムの橋渡しができるようになった。
※実際は、ユーザーがイテレータを意識することはない。

for_eachの実装例

```
template <class First, class Last, class F>
void for_each_impl(First first, Last last, F f, mpl::true_) {}

template <class First, class Last, class F>
void for_each_impl(First first, Last last, F f, mpl::false_)
{
    f(deref(first));
    for_each_impl(next(first), last, f,
        result_of::equal_to<
            typename result_of::next<First>::type, Last>());
}

template <class Seq, class F>
void for_each(const Seq& seq, F f)
{
    for_each_impl(begin(seq), end(seq), f,
        result_of::equal_to<typename result_of::begin<Seq>::type,
            typename result_of::end<Seq>::type >());
}
```

毎回型が変わるので、ループではなく再帰で書く

- シーケンスに対するSTLライクなアルゴリズムが提供されている。
- Output Iteratorではなく戻り値で返す。
- Viewを用いた遅延評価が特徴。複数のアルゴリズムの適用を一度のループで処理する。
- アルゴリズムは「関数」と「メタ関数」、実行時とコンパイル時で一樣なものが提供される。
実行時アルゴリズムはboost::fusion名前空間。
コンパイル時アルゴリズムはboost::fusion::result_of名前空間。

アルゴリズムの適用結果は、アルゴリズムを適用した型が返される。
そのため、実行時の値に対するアルゴリズムだけではなく、
戻り値のために、コンパイル時の型に対するアルゴリズムが用意されている。

```
typedef fusion::vector<int, char, std::string> vector_t;  
const vector_t v(1, 'a', "Hello");  
  
typedef  
    fusion::result_of::transform<const vector_t, to_string>::type  
result_type;  
  
const result_type result = fusion::transform(v, to_string());
```

transformのような処理は、ユーザーコードで書くのは稀。
こういった処理はライブラリコードの関数テンプレートで行うのが一般的。

関数	作用	説明
fold	$f(\dots f(f(\text{initial_state}, e_1), e_2) \dots e_N)$	前から畳み込む。
reverse_fold	$f(\dots f(f(\text{initial_state}, e_N), e_{N-1}) \dots e_1)$	後ろから畳み込む。
iter_fold	$f(\dots f(f(\text{initial_state}, it_1), it_2) \dots it_N)$	要素ではなくイテレータが渡されるfold
reverse_iter_fold	$f(\dots f(f(\text{initial_state}, it_N), it_{N-1}) \dots it_1)$	要素ではなくイテレータが渡されるreverse_fold
accumulate	$f(\dots f(f(\text{initial_state}, e_1), e_2) \dots e_N)$	foldと同じ。
for_each	$f(e) \dots$	全ての要素に関数を適用

関数	説明
any	述語を満たす要素があるか
all	全ての要素が述語を満たすか
none	述語を満たす要素が存在しないか
find	値を検索
find_if	述語による検索
count	指定された値の要素を数える
count_if	指定された述語を満たす要素を数える

関数	説明
filter	指定された型のみを抽出
filter_if	述語を満たす要素を抽出
transform	全ての要素に変換関数を適用
replace	値を置き換える
replace_if	述語を満たす要素を置き換える
remove	指定された型を削除
remove_if	述語を満たす要素を削除
reverse	シーケンスを逆順にする
clear	空のシーケンスを返す
erase	イテレータによる要素削除
erase_key	キーの指定による要素削除

関数	説明
insert	指定位置に要素を挿入
insert_range	指定位置にシーケンスを挿入
join	2つのシーケンスを連結
zip	複数のシーケンスを綴じ合わせる
pop_back	最後尾要素を削除
pop_front	先頭要素を削除
push_back	最後尾に要素を追加
push_front	先頭に要素を追加

Fusionには、ユーザー定義型をFusionシーケンスにアダプトする機構が用意されている。以下はユーザー定義型のメンバ変数を列挙する処理:

```
struct Person {  
    int identifier;  
    std::string name;  
    int age;  
};
```

```
BOOST_FUSION_ADAPT_STRUCT(  
    Person,  
    (int, identifier)  
    (std::string, name)  
    (int, age)  
)
```



```
1  
Akira  
25
```

```
const Person person = {1, "Akira", 25};  
fusion::for_each(person, std::cout << _1 << ' ');
```

Chapter.03

***Fusion*はどこで使うのか**

Boost.Fusionの使いどころは大きく2つ:

1. 名前が付いているがシーケンスとしても扱いたい場合 (RGBなど)
2. DSELの内部実装(Boost.Spirit.Qi/Karma)
3. Fusion Sequenceをコンセプトとするライブラリへの一括アダプト(Boost.Geometry)

RGB値は、構造体として扱うと、名前を付けられるがシーケンスとして扱えず、配列として扱うと名前が...
という一長一短の設計の選択肢がある。

RGBを構造体にしてFusionシーケンスにアダプトすることで、名前でのアクセスと、名前を必要としないシーケンスとしてのアクセス両方が手に入る。

以下は、簡単な画像処理(ネガ反転)。

OpenCVは内部の要素型を外部から指定できるので、Fusionシーケンスへのアダプトが簡単にできる。

```
struct Color {  
    uchar r, g, b;  
};  
...  
  
Color c;  
fusion::for_each(c, _1 = 255 - _1);
```



- DSELでは、異なる型のシーケンスを扱う機会が多い。
たとえば、正規表現やパーサーコンビネータ。

これらの内部実装にBoost.Fusionを使用することで、
ユーザーコードが簡潔で柔軟になる。

Boost.SpiritではFusionを、パース式、およびパース結果の型として使用する。

```
fusion::vector<int, char, double> result;  
parse("1 a 3.14", int_ >> char_ >> double_, result);  
  
std::cout << result << std::endl;
```

(1 a 3.14)

Fusionシーケンスで結果を返すことにより、
BOOST_FUSION_ADAPT_STRUCTでアダプトされた
ユーザー定義型へ一発変換できる。

```
struct X {  
    int n;  
    char c;  
    double d;  
};  
...  
  
X result;  
parse("1 a 3.14", int_ >> char_ >> double_, result);  
  
std::cout << result.n << ' ' << result.c << ' ' << result.d;
```

1 a 3.14

さらに、charのシーケンスを以下のいずれの型でも扱えるため、ユーザーコードが非常に柔軟になる:

fusion::vector<char, char, ...>

std::string

std::vector<char>

```
fusion::vector<char, char, char> result;  
parse("1 a 3.14", char_ >> char_ >> char_, result);
```

<http://ideone.com/NTZ6z>

```
std::string result;  
parse("1 a 3.14", char_ >> char_ >> char_, result);
```

<http://ideone.com/4FonC>

```
std::vector<char> result;  
parse("1 a 3.14", char_ >> char_ >> char_, result);
```

<http://ideone.com/do3lO>

Boost.Geometryでは、Fusionシーケンスとしてアダプトされた全ての型を、Geometryのcoordinateとして扱うことができる。

以下は、ユーザー定義型で、2つの点の距離を求めるdistanceアルゴリズムを使用する例：

```
namespace bg = boost::geometry;

struct Point { float x, y; };
BOOST_FUSION_ADAPT_STRUCT(Point, (float, x) (float, y))

const Point a = {0.0f, 0.0f};
const Point b = {3.0f, 3.0f};

std::cout << bg::distance(a, b) << std::endl;
```

4.24264

この機構はワシが作った。

Chapter.04

ライフライ設計のお話

- タプルは値(メンバ変数)と値の集合(クラス)に、名前のない複合データ型である。
- Boost.Fusionでは、ユーザー定義型をFusionシーケンスにアダプトすることによって、名前ありタプルと見なすことができるようになる。
つまり、以下のようになる:

タプル: 名無しユーザー定義型

ユーザー定義型: 名前ありタプル

- Boost.Fusionでは、「名前」を意識して、ライブラリコードとユーザーコードで、コードの棲み分けを行うことが重要。
何も考えずユーザーコードをFusionを使いまくると、名前のない値で溢れかえってしまう。以下のようにしよう:

ユーザーコード：名前あり世界

(ユーザー定義型 + アダプト) x アルゴリズム

ライブラリコード：名無し世界

fusion::vector, transform, etc...

- タプルは、クラスを作るのがめんどくさいときに即興で使われることが多い。
- しかし、やはりめんどくさがらずに値(メンバ変数)と値の集合(クラス)には名前を付けよう。
- データを単なるシーケンスとして扱っていいのはライブラリの中だけ。

- ユーザーコードでは、ユーザー定義型をFusionシーケンスへアダプトすることで、有用なアルゴリズムを手に入れよう。
- ライブラリがFusionで設計されてさえいれば、ユーザー定義型ですでに定義済みの有用なアルゴリズムが手に入る (パーサー、一般的なアルゴリズム、幾何学の関数、線形代数の関数など)。

全てのライブラリコードで名前が使えないと少々不便。
そんなときは、型と値の対応表であるfusion::mapを使用する。

```
struct id {}; struct name {};

template <class AssocSeq>
void foo_impl(const AssocSeq& seq)
{
    std::cout << "id:"    << fusion::at_key<id>(seq)
               << " name:" << fusion::at_key<name>(seq);
}

template <class Seq>
void foo(const Seq& seq)
{
    foo_impl(fusion::map_tie<id, name>(
        fusion::at_c<0>(seq),
        fusion::at_c<1>(seq)));
}

foo(fusion::make_vector(1, "Akira"));
```

ライブラリ世界での名前

- fusion::mapを使用することで、ライブラリ世界で名前が手に入る。

ユーザーコード：名前あり世界

(ユーザー定義型 + アダプト) x アルゴリズム

名前

fusion::map

ライブラリコード：名無し世界
fusion::vector, transform, etc...

- Boost.Fusionはタプルをリストと見なす
- Boost.Fusionはタプルに名前をもたらす
- ライブラリの設計にBoost.Fusionを取り入れることで、ユーザーコードが柔軟になる
- まだまだ事例が少ないので、各自で応用を考えよう

- Boost Fusion Library
<http://www.boost.org/libs/fusion/doc/html/index.html>
- Fusion by example
http://www.boostcon.com/site-media/var/sphene/sphwiki/attachment/2007/05/28/An_Introduction_to_Boost.Fusion.pdf
- cpppeg : PEGパーサー
<https://github.com/kik/cpppeg/blob/master/peg.hpp>
- Spirit: History and Evolution
<http://blip.tv/file/4245756>