Output Iteratorの置き換えと Boost.Rangeの拡張

高橋 晶(Akira Takahashi)
株式会社ロングゲート
@cpp_akira

2012/05/26(土) Boost.勉強会 #9 つくば

Boost C++ Libraries

この発表について

C++Now! 2012のLibrary in a Weekで
発表した内容の日本語版です。

Boost C++ Libraries

Library in a Week

- 一週間のC++Now!の中で、みんなでライブラリを書こう！というプロジェクト
- 2012年のテーマは「C++11時代のアルゴリズム」
  - Boost.RangeやBoost.Algorithm、ASL (Adobe Source Library)の拡張が盛んに行われた
- Library in a Weekは5日間、毎朝08:00から09:00まで行われる。
  - 毎朝1時間みんなで集まって作業するのではなく、セッションの合間にある休み時間等で作業する
  - C++Now!は毎日08:00から22:00まで。
  - 時間ないです。
- そんなプロジェクトで、私が構想しているアイデアと、現在進めているプロジェクトの発表をしてきました。

1st

Output Iterators Must Go

Boost C++ Libraries

このアイデアの概要

- C++11となった今となっては、Output Iteratorは必要ない。
- なぜなら、C++11にはラムダ式があるから。
- いくつかのSTLアルゴリズムは、Output IteratorをUnaryFunctionに置き換えることができる。

Boost C++ Libraries

基本的な例：std::copy

std::copyはstd::for_each + ラムダで置き換えることができる。

Before:

```
std::vector<int> v = {1, 2, 3};
std::vector<int> result;

std::copy(v.begin(), v.end(), std::back_inserter(result));
```

After:

```
std::vector<int> v = {1, 2, 3};
std::vector<int> result;

std::for_each(v.begin(), v.end(),
                    [&](int x) { result.push_back(x); });
```

この置き換えはそこそこ便利

より実用的な例：集合演算

集合演算のアルゴリズムは、Output Iteratorバージョンしか用意されていないため、カスタム操作がとても書きにくい。

現在の集合演算アルゴリズム

```
std::set<int> a = {1, 2, 3};
std::set<int> b = {4, 5, 6};
std::set<int> result;

std::set_union(a.begin(), a.end(),
               b.begin(), b.end(),
               std::inserter(result, result.end()));
```

Insert Iteratorアダプタは全然便利じゃない。

より実用的な例：集合演算

集合演算のアルゴリズムは、Output Iteratorバージョンしか用意されていないため、カスタム操作がとても書きにくい。

新たなの集合演算アルゴリズムの提案

```
std::set<int> a = {1, 2, 3};
std::set<int> b = {4, 5, 6};
std::set<int> result;

set_union(a.begin(), a.end(),
          b.begin(), b.end(),
          [](int x) { result.insert(x); });
```

Output IteratorをUnaryFunctionに置き換えた。
これは実際とても便利で、カスタム操作も簡単に書ける。

実装の話 1/4

- 基本的な実装はとても簡単。
- 関数呼び出しを行うOutput Iteratorを書いてラップすればいい。そのようなOutput Iteratorはすでにboost::function_output_iteratorとして用意されている。

```
template <class InputIterator1, class InputIterator2, class UnaryFunction>
void set_union(InputIterator1 first1, InputIterator1 last1,
               InputIterator2 first2, InputIterator2 last2,
               UnaryFunction&& f)
{
    std::set_union(
            first1, last1, first2, last2,
            boost::make_function_output_iterator(boost::move(f)));
}
```

実装の話 2/4

- function_output_iteratorの中身

```
template <class UnaryFunction>
struct function_output_iterator {
  explicit function_output_iterator(const UnaryFunction& f) : m_f(f) {}

  struct output_proxy {
    output_proxy(UnaryFunction& f) : m_f(f) { }

    template <class T> output_proxy& operator=(const T& value) {
      m_f(value);
      return *this;
    }
    UnaryFunction& m_f;
  };

  output_proxy operator*() { return output_proxy(m_f); }
  function_output_iterator& operator++() { return *this; }
  function_output_iterator& operator++(int) { return *this; }
  UnaryFunction m_f;
};
```

実装の話 3/4

- std名前空間に同じ名前のset_union()関数を共存させる方法
- 型TがUnaryFunctionかどうかを判定するis_unary_callableメタ関数を作ってSFINAEする。

```
template <class InputIterator1, class InputIterator2, class UnaryFunction>
auto set_union(InputIterator1 first1, InputIterator1 last1,
               InputIterator2 first2, InputIterator2 last2,
               UnaryFunction&& f) ->
        typename boost::enable_if<is_unary_callable<
                                  UnaryFunction,
                                  decltype(*first1)
        >>::type
{
    std::set_union(
            first1, last1, first2, last2,
            boost::make_function_output_iterator(boost::move(f)));
}
```

これで、テンプレートパラメータUnaryFunctionが単項関数呼び出し可能でなければ、この関数はオーバーロード解決から除外される。

Boost C++ Libraries

実装の話 4/4

- is_unary_callableは、C++11のdecltype + SFINAEで、関数呼び出しの式が正当かどうかをチェックし、正当であればtrue_type、不正であればfalse_typeを返すようにしてる。

```
template <class F, class V>
struct is_unary_callable_base {
private:
  template <class F2, class V2>
  static auto check(F2&& f, V2 v) -> decltype((f(v)), std::true_type());

  static auto check(...) -> std::false_type;
public:
  typedef
    decltype(check(std::declval<F>(), std::declval<V>()))
  type;
};

template <class F, class V>
struct is_unary_callable : is_unary_callable_base<F, V>::type {};
```

Output Iteratorの置き換えまとめ

- Output Iteratorは、ラムダ式のあるC++11時代ではUnaryFunctionでの置き換えが十分に便利。

- この実装はそのうちドキュメントとテストを書いてBoost.Algorithmに提案する予定。
(Library in a Weekの成果はその年のうちに反映させる、という方針のため)

- 実装はこちら:
https://github.com/faithandbrave/Set-Algorithm

2nd

OvenToBoost プロジェクト

OvenToBoostプロジェクトとは

- OvenというのはP-Stade C++ Librariesに含まれるRangeライブラリ
- OvenはBoost.Rangeよりもいろいろと揃ってて便利
- 現在、OvenをBoost.Rangeの拡張として移植するプロジェクトを進めています。
  https://github.com/faithandbrave/OvenToBoost

Boost.Range概要

- イテレータの組をとるSTLアルゴリズムのラッパー、Rangeアルゴリズムが提供されている
- それに加えて、遅延評価のリスト操作のためのRangeアダプタと呼ばれる機能が提供されている。

```
const std::vector<int> v = {3, 1, 4, 2, 5};

boost::for_each(v | filtered(is_even), print); // 偶数値を出力
```

```
4
2
```

- Rangeアダプタの適用にはoperator|()を使用する。これはUNIXのパイプにあやかっている。
- Rangeアダプタの名前は過去分詞を使用している。
- このサンプルにおいてfor_each + filteredは1ループで処理される

Boost.Rangeの問題点

- Rangeアダプタが少なすぎる
  - takenがない
  - droppedもない
  - 無限Rangeがない
  - 足りなすぎる・・・。
- Boost.RangeのRangeアダプタはラムダを扱えない
- Ovenはこれらの問題に対する解決策を持っている

Boost C++ Libraries

taken Rangeアダプタ

takenはRangeから先頭N個の要素を取り出したRangeを生成する

```
const std::vector<int> v = {3, 1, 4, 2, 5};

boost::for_each(v | taken(2), print);
```

```
3
1
```

dropped Rangeアダプタ

droppedはRangeから先頭N個の要素を除いたRangeを生成する

```
const std::vector<int> v = {3, 1, 4, 2, 5};

boost::for_each(v | dropped(2), print);
```

```
4
2
5
```

elements Rangeアダプタ

elementsはRangeのオブジェクトから特定要素のみを抽出する

```
struct Person {
  int id;
  std::string name;
  …
};
BOOST_FUSION_ADAPT_STRUCT(…)

const std::vector<Person> v = {
    {1, "Alice"}
    {2, "Carol"}
    {3, "Bob"}
 };

boost::for_each(v | elements<1>(), print);
```

```
Alice,Carol,Bob
```

elements_key Rangeアダプタ

elements_keyはタグを使用してRangeのオブジェクトから特定要素のみを抽出する

```
struct id_tag {}; struct name_tag {};

struct Person {
  int id;
  std::string name;
  …
};
BOOST_FUSION_ADAPT_ASSOC_STRUCT(…)

const std::vector<Person> v = {
    {1, "Alice"}
    {2, "Carol"}
    {3, "Bob"}
 };

boost::for_each(v | elements_key<name_tag>(), print);
```

```
Alice,Carol,Bob
```

Boost C++ Libraries

iteration関数

iteration()は無限Rangeを生成する関数。第2引数は次の値を計算する関数。

```
int next(int x) { return x * 2; }

boost::for_each(iteration(1, next) | taken(5), print);
```

```
1
2
4
8
16
```

regular関数

regular()関数は関数オブジェクトをRegular Concept(DefaultConstructible & CopyAssinable)を満たすように変換する。ラムダはRegular Conceptを満たさないので、イテレータに入れるとInput Iteratorになれない

```
template <class InputIterator, class F>
F for_each_(InputIterator first, InputIterator last, F f) {
  InputIterator it; // default construct
  it = first; // copy assign

  while (it != last) { f(*it); ++i; }
  return f;
}

template <class Range, class F>
F for_each_(const Range& r, F f)
{ return for_each(boost::begin(r), boost::end(r), f); }

using boost::lambda::_1;
for_each_(r | filtered(_1 % 2 == 0), f);          // Error!
for_each_(r | filtered(regular(_1 % 2 == 0)), f); // OK
```

regular演算子operator|+()

regular()関数のシンタックスシュガー。
operator|()と同じ優先順と結合規則を持つ演算子が存在しなかったので、
operator|()と単項のoperator+()を組み合わせた複合演算子を作った。

```
template <class InputIterator, class F>
F for_each_(InputIterator first, InputIterator last, F f) {
  InputIterator it; // default construct
  it = first; // copy assign

  while (it != last) { f(*it); ++i; }
  return f;
}

template <class Range, class F>
F for_each_(const Range& r, F f)
{ return for_each(boost::begin(r), boost::end(r), f); }

using boost::lambda::_1;
for_each_(r |  filtered(_1 % 2 == 0), f); // Error!
for_each_(r |+ filtered(_1 % 2 == 0), f); // OK
```

Ovenの機能を組み合わせた例：素数列

エラトステネスのふるいを使った素数の無限数列。

```
range sieve(range r)
{
  return r | dropped(1) |+ filtered(_1 % value_front(r) != 0);
}

range primes =
    iteration(range(
      iteration(2, regular(_1 + 1))), sieve) |
transformed(value_front);


for_each(primes, print);
```

```
2 3 5 7 11 …
```

OvenToBoostの現状

- 最優先の機能は実装完了。
- テストも完了。
- ドキュメントが終わっていなかったが、Library in a Weekでほぼ完了。
  - 現在、zakさんに英語を綺麗にしてもらって、hotwatermorningさんにドキュメントのビルドを手伝ってもらってる
- ドキュメント整備がまもなく完了するので、近々Boostのレビューリクエストを提出する予定。