# Boostのあるプログラミング生活

高橋晶 (Akira Takahashi)

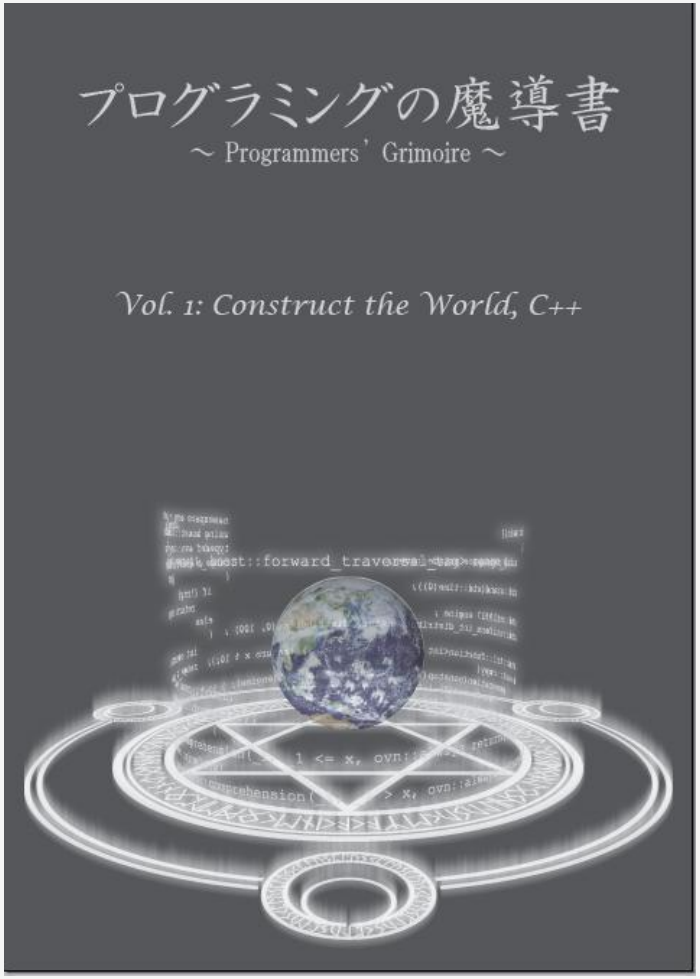
ブログ:「Faith and Brave - C++で遊ぼう」
http://d.hatena.ne.jp/faith_and_brave/

わんくま横浜勉強会 2010/08/28(土)

Boost C++ Libraries

# 宣伝コーナー

プログラミング雑誌
(ライクな書籍)を始めました。




タイトルは

プログラミングの魔導書
～Programmers' Grimoire～

です。

書籍版は完全受注生産のためすでに締め切りましたが、PDF版は購入可能です。
http://longgate.co.jp/products.html

Boost C++ Libraries

# はじめに

Boostによって、C++でのプログラミングは
かなり容易で、強力なものとなりました。

今回は、Boost 1.43.0時点でのとくに強力で、
私が好んで使用しているライブラリを紹介します。

Boost C++ Libraries

# アジェンダ

1. MultiIndex
2. Spirit.Qi
3. PropertyTree
4. Range 2.0
5. Under Construction

Chapter 1

# Multi Index

統合データ構造

Boost C++ Libraries

# MultiIndexとは何か

複数のインデックスを持てるコンテナ！

組み合わせは自由自在！

# Boost C++ Libraries MultiIndex Introduction 1/5

まずは簡単なサンプルから。

挿入順を知っているstd::setです。

```
using namespace boost::multi_index;

typedef multi_index_container<
  int,
  indexed_by<
    ordered_unique<identity<int> >, // std::setで、
    sequenced<>                     // 挿入順を知ってる
  >
> container;
```

※ファイルに入っている順序に一覧表示、という処理を想定

Next >>

# Boost C++ Libraries MultiIndex Introduction 2/5

何も気にしなければstd::setとして使えます

```
container c;

c.insert(3);
c.insert(1);
c.insert(4);

boost::for_each(c, disp);
```

```
1 3 4
```

std::setと同じく、ソートされて表示される

Next >>

# MultiIndex Introduction 3/5

1番目のインデックス(つまりsequenced)として
使用すると挿入順に処理できます

```
container c;

c.insert(3);
c.insert(1);
c.insert(4);

boost::for_each(c.get<1>(), disp);
```

```
3 1 4
```

挿入順に出力される。
ordered_uniqueを明示的に取得したい場合はget<0>()

Next >>

# Boost C++ Libraries MultiIndex Introduction 4/5

0とか1とかいうインデックス番号なんて覚えられない！
インデックスにタグ(名前)を付けましょう。

```
struct order {};
struct seq {};

typedef multi_index_container<
  int,
  indexed_by<
    ordered_unique<tag<order>, identity<int> >,
    sequenced<tag<seq> >
  >
> container;
```

Next >>

# Boost C++ Libraries MultiIndex Introduction 5/5

タグを指定してインデックスを取得できます。

```
container c;

c.insert(3);
c.insert(1);
c.insert(4);

boost::for_each(c.get<seq>(), disp);
```

```
3 1 4
```

挿入順に出力される。
ordered_uniqueを明示的に取得したい場合はget<order>()

Boost C++ Libraries

# End of Introduction

あとはインデックスの種類と組み合わせだけ！

Boost C++ Libraries

# 提供されているインデックス

| インデックス | 対応するコンテナ |
| --- | --- |
| ordered_unique | std::set (デフォルト) |
| ordered_non_unique | std::multiset |
| hashed_unique | boost::unordered_set |
| hashed_non_unique | boost::unordered_multiset |
| sequenced | std::list |
| random_access | std::vector |

Boost C++ Libraries

# 活用例 1/4

## 複数のキーを持つ辞書

| 名前 | サイズ | 種類 | 更新日時 |
|---|---|---|---|
| boost | | ファイル フォルダ | 2010/04/26 18:17 |
| doc | | ファイル フォルダ | 2010/04/26 18:28 |
| libs | | ファイル フォルダ | 2010/04/26 18:27 |
| more | | ファイル フォルダ | 2010/04/26 18:28 |
| people | | ファイル フォルダ | 2010/04/26 18:14 |
| status | | ファイル フォルダ | 2010/04/26 18:14 |
| tools | | ファイル フォルダ | 2010/04/26 18:28 |
| wiki | | ファイル フォルダ | 2010/04/26 18:28 |
| boost.css | 2 KB | Cascading Style Sh... | 2010/04/26 18:28 |
| boost.png | 7 KB | PNG イメージ | 2010/04/26 18:28 |
| boost-build.jam | 1 KB | JAM ファイル | 2010/04/26 18:28 |
| bootstrap.bat | 3 KB | MS-DOS バッチ ファ... | 2010/04/26 18:28 |
| bootstrap.sh | 11 KB | SH ファイル | 2010/04/26 18:28 |
| index.htm | 1 KB | Opera Web Docume... | 2010/04/26 18:28 |
| index.html | 6 KB | Opera Web Docume... | 2010/04/26 18:28 |
| INSTALL | 1 KB | ファイル | 2010/04/26 18:28 |
| Jamroot | 26 KB | ファイル | 2010/04/26 18:28 |
| LICENSE_1_0.txt | 2 KB | テキスト文書 | 2010/04/26 18:28 |
| rst.css | 3 KB | Cascading Style Sh... | 2010/04/26 18:28 |

Next >>

Boost C++ Libraries

# 活用例 2/4

こんな構造体があるとして

```
struct file {
    std::string name;
    std::size_t size;
    file_kind   kind;
    datetime    update_date;
};
```

Next >>

Boost C++ Libraries

# 活用例 3/4

memberを使用して、
どのメンバ変数をキーにするか指定する

```
typedef multi_index_container<
  file,
  indexed_by<
    ordered_unique<    member<file, std::string, &file::name> >,
    ordered_non_unique<member<file, std::size_t, &file::size> >,
    ordered_non_unique<member<file, file_kind,   &file::kind> >,
    ordered_non_unique<member<file, datetime,    &file::update_date> >
  >
> file_container;
```

Next >>

# 活用例 4/4

タグを付けて、最終的にこんな感じで使えます

```
void disp_file_info(const file&);


file_container files;
...

// 名前順に並び替えて表示
boost::for_each(files.get<name_order>(), disp_file_info);

// ファイル種類をキーにして、jpgファイルを検索
if (files.get<kind_order>().find(file_kind::jpeg) !=
    files.get<kind_order>().end()) {
    ...
}
```

Boost C++ Libraries

# まとめ

- 応用範囲はとても広い
- 実装の話はしませんが、効率は自分で作るより数万倍いいです。

  そのへんの話はk.inabaさんのセッションをみてください:
  http://www.ustream.tv/recorded/2968920
- ドキュメント
  http://www.boost.org/libs/multi_index/doc/index.html

Boost C++ Libraries

# アドバイス

分割統治しましょう。

じゃないとコード量が爆発します。

Typedef Templateとか覚えておくといいです。

Chapter 2

# Spirit.Qi

構文解析DSEL

Boost C++ Libraries

# はじめに

自前の構文解析してたりしませんか？

Boost C++ Libraries

# はじめに

文字列のfind関数とか使ってませんか？

Boost C++ Libraries

# はじめに

1文字ずつ自分で解析したりしてませんか？

Boost C++ Libraries

# はじめに

そんな自前の構文解析から卒業できます。

そう、Qiならね。

Boost C++ Libraries

# Qiとは何か

- Boost.Spiritの構文解析ライブラリであり、DSEL(Domain Specific Enbedded Language : ドメイン特化組み込み言語)

- Qiの読みは「キ」。気です。
  私は「キューアイ」と読んでるのでそれでいきます。

Boost C++ Libraries

# まず下準備

こんなの用意しておきます。

begin(), end()を書かないための単なるラッパーです。

```
template <class Parser, class Result>
bool parse(const std::string& s, const Parser& p, Result& result)
{
    std::string::const_iterator it = s.begin();
    return boost::spirit::qi::parse(it, s.end(), p, result);
}
```

ついでに、名前空間を省略しておきます。

```
using namespace boost::spirit::qi;
```

# Boost C++ Libraries カッコの中身を取り出してみる

operator>>()演算子と、
int_, char_等の各型用のパーサーを使用します。

```
const std::string s =  “(123)” ;

typedef int result_type;
typedef rule<std::string::const_iterator, result_type()> rule_type;

const rule_type r =  ‘(’  >> int_ >>  ‘)’ ;

result_type result;
parse(s, r, result);

std::cout << result << std::endl;
```

```
123
```

処理は実質1行！

Boost C++ Libraries

# データ型パーサー

| パーサー | 型(マッチする例) |
| --- | --- |
| int_ | int(123) |
| double_ | double(3.14) |
| bin | unsigned int(0101) |
| hex | unsigned int(1a) |
| char_ | char(a) |
| auto_ | 自動的に推論 |

Boost C++ Libraries

# CSVを解析してみる

## %はstd::vector<T>を返す単純な繰り返し用パーサー

```
const std::string s =  “123,456,789” ;

typedef std::vector<int> result_type;
typedef rule<std::string::const_iterator, result_type()> rule_type;

const rule_type r = int_ %  ‘,’ ; // カンマ区切り、要素はint

result_type result;
parse(s, r, result);

boost::for_each(result, disp);
```

```
123 456 789
```

# IPv4を解析してみる

,を.に変えるだけ。

```
const std::string s =  “192.168.0.1” ;

typedef std::vector<int> result_type;
typedef rule<std::string::const_iterator, result_type()> rule_type;

const rule_type r = int_ %  ‘.’ ; // ドット区切り、要素はint

result_type result;
parse(s, r, result);

boost::for_each(result, disp);
```

```
192 168 0 1
```

# 型が異なるシーケンスの場合

## Boost.Fusionのシーケンスで結果を受け取る

```
const std::string s =  “123,Akira,25” ;

typedef fusion::vector<int, std::string, int> result_type;
typedef rule<std::string::const_iterator, result_type()> rule_type;

const rule_type r =
          int_ >>  ‘,’  >> *(char_ -  ‘,’ ) >>  ‘,’  >> int_;

result_type result;
parse(s, r, result);

std::cout << result << std::endl;
```

```
(123 Akira 25)
```

Boost C++ Libraries

# パーサー演算子

| 演算子 | 説明 |
| --- | --- |
| >> | シーケンス。<br>Fusionのシーケンスを返す。 |
| *a | 0回以上の繰り返し。<br>std::vector<A>, std::stringを返す。 |
| a % b | b区切りの1回以上の繰り返し。<br>std::vector<A>を返す。 |
| a – b | 差。bではない間。 |

Boost C++ Libraries

# パーサーディレクティブ

| 演算子 | 説明 |
| --- | --- |
| omit[a] | aのパースは行うが結果はいらない。 |
| repeat[a] | パースに成功する限り、aパーサーを繰り返す。<br>std::vector<A>を返す。 |
| repeat(N)[a] | N回aパーサーを繰り返す。<br>std::vector<A>を返す。 |

# Boost C++ Libraries パーサーディレクティブの例 1/2

カッコの中身を取り出す際、
前に付いている文字列はいらない、という場合。

```
const std::string s =  “abc(123)” ;

typedef int result_type;
typedef rule<std::string::const_iterator, result_type()> rule_type;

const rule_type r =
           omit[*(char_ - '(')] >>  ‘(’  >> int_ >>  ‘)’ ;

result_type result;
parse(s, r, result);

std::cout << result << std::endl;
```

```
123
```

# パーサーディレクティブの例 1/2

N個のデータを読む。

```
const std::string s =  “(123)(456)(789)” ;

typedef std::vector<int> result_type;
typedef rule<std::string::const_iterator, result_type()> rule_type;

const rule_type r =
           repeat(3)[ ‘(’  >> int_ >>  ‘)’ ]; // (3)は省略可

result_type result;
parse(s, r, result);

boost::for_each(result, disp)
```

```
123 456 789
```

# 最後の例 1/2

パース時に構造体に一発変換する

```
struct person {
    int id;
    std::string name;
    int age;
};

BOOST_FUSION_ADAPT_STRUCT(
    person,
    (int, id)
    (std::string, name)
    (int, age)
)
```

Boost.Fusionには、ユーザー定義の型を
Fusionのシーケンスとしてアダプトする機能がある

# 最後の例 2/2

## 構造体で結果を受け取る

```
const std::string s = “123,Akira,25” ;

typedef person result_type;
typedef rule<std::string::const_iterator, result_type()> rule_type;

const rule_type r =
         int_ >> ‘,’ >> *(char_ - ‘,’ ) >> ‘,’ >> int_;

result_type p;
parse(s, r, p);

std::cout << p.id << “ ” << p.name << “ ”
          << p.age << std::endl;
```

```
123 Akira 25
```

Boost C++ Libraries

# 話さなかったこと

- セマンティックアクション
- Spirit.Phoenixとの連携
- 文字コードの扱いほげほげ
- イテレータによる遅延パース
- エラー処理

Boost C++ Libraries

# まとめ

- Spirit.Qiは簡単かつ強力です。
- 構造体へも一発パースできます。
- 読めます。

- ドキュメント
  http://www.boost.org/libs/spirit/doc/html/spirit/qi.html

Chapter 3

# Property Tree

ツリー構造の汎用プロパティ管理

Boost C++ Libraries

# PropertyTreeとは

- ツリー構造への一般的なアクセス方法を提供するライブラリ
- 使用する型はboost::property_tree::ptree
- 待望のXMLパーサー、JSONパーサー搭載
  (他にも、INFOとINIのパーサーがある)

Boost C++ Libraries

# XMLパーサー

XMLの読込、要素、属性の取得。
XMLパーサーにはRapidXmlを採用している。

```
<root>
  <elem attr="World">
    Hello
  </elem>
</root>
```

```
using namespace boost::property_tree;

ptree pt;
read_xml("test.xml", pt, xml_parser::trim_whitespace);

// 要素の取得
const string& elem = pt.get<string>("root.elem");

// 属性の取得 : <xmlattr>という特殊な要素名を介してアクセスする
const string& attr = pt.get<string>("root.elem.<xmlattr>.attr");
```

Boost C++ Libraries

# JSONパーサー

JSONの読込、データの取得。

パーサーはSpirit.Classicで書かれている。

```
{
  "Data": {
    "Value": 314,
    "Str": "Hello"
  }
}
```

```
using namespace boost::property_tree;

ptree pt;
read_json("test.json", pt);

const int     value = pt.get<int>("Data.Value");
const string& str   = pt.get<string>("Data.Str");
```

# Boost C++ Libraries さらっとまとめと注意事項

- XMLパーサーがやっと入りました。
- XPathじゃなくPropertyTreeの独自構文によるアクセスなので注意。
- ptree::getは失敗時に例外投げる。
  boost::optionalを返すptree::get_optionalもあります。
- ドキュメント
  http://www.boost.org/doc/html/property_tree.html

Chapter 4

# Range 2.0

## Adaptor & Algorithms

# Boost.Range 2.0

Boost 1.43.0で、
Boost.Rangeのバージョンが2.0になりました。

# Boost.Range 1.0

1.0でできること：

```
template <class Range, class T>
typename boost::range_iterator<Range>::type
  find(Range& r, const T& value)
{
  return std::find(boost::begin(r), boost::end(r), value);
}
```

```
int ar[3];
int* p = find(ar, 1);

std::vector<int> v;
std::vector<int>::iterator it = find(v, 1);
```

配列やコンテナを同じように扱うための
ユーティリティが提供されていた。

# Rangeアルゴリズム

2.0では、前項のような
Rangeに対するアルゴリズムが提供されている。

```
const std::vector<int> v = {3, 1, 4};
boost::sort(v);
```

```
const std::vector<int> v = {1, 2, 3};
boost::for_each(v, [](int x) { std::cout << x << std::endl; });
```

```
const std::vector<int> v  = {1, 2, 3};
const std::list<int>   ls = {1, 2, 3};
const bool is_same = boost::equal(v, ls);
```

begin()/end()を書かなくてもよくなった。

Boost C++ Libraries

# Rangeアダプタ

Rangeに対する複数の操作を合成する
「Rangeアダプタ」も少しだが提供されている。

```
using namespace boost::adaptors;
```

```
const std::vector<int> v = {1, 2, 3, 4, 5};
boost::for_each(v | filtered(is_even) | transformed(to_string),
                [](const std::string&) { ... });
```

```
const std::map<int, std::string> m;
boost::for_each(m | map_values, [](const std::string&) { ... });
```

```
const std::vector<std::shared_ptr<X>> v;
boost::for_each(v | indirected, [](const X&) { ... });
```

Rangeアダプタの適用には`operator|()`を使用する。

Boost C++ Libraries

# Rangeアダプタの種類

| アダプタ | 効果 |
| --- | --- |
| r \| filtered(pred) | pred(x) == trueとなる要素を抽出する |
| r \| transformed(f) | 各要素にf(x)を適用する |
| r \| map_keys<br>r \| map_values | map<T1, T2>, vector<pair<T1, T2> >のようなRangeに対し、map_keysはT1を抽出し、map_valuesはT2を抽出する |
| r \| indirected | スマートポインタのRangeへの要素アクセスを、*pではなくxでアクセス可能にする |
| erc... | |

Boost C++ Libraries

# 遅延評価

```
r | filtered(pred) | transformed(f);
```

は、この時点ではRangeの全要素の横断は行われない。

Rangeアダプタは、各要素へのイテレータをラップするような
実装になっているため、以下のような

```
for_each(r | filtered(pred) | transformed(f), ...);
```

アルゴリズムによるイテレータに対する評価が行われたときに

filtered : 要素の読み飛ばし
transformed : 要素への関数適用

が行われる。
そのため、このRangeに対する複数の操作は一度のループで
実行される。

Boost C++ Libraries

# Rangeまとめ

- Range Algorithmで、STLアルゴリズムがより使いやすくなりました。
- Rangeアダプタで、より抽象的なプログラミングが可能になり、C++で遅延評価のリスト処理ができるようになりました。
- でもまだまだ、Rangeアダプタが足りない。Ovenが強力です。
- ドキュメント
  http://www.boost.org/libs/range/doc/html/index.html

Final Chapter

# Under Construction

Boost C++ Libraries

# 未来のお話

Boost候補として開発されているライブラリを
いくつか紹介します。

1. Mirror
2. STM
3. LA

Boost C++ Libraries

# Mirror

リフレクションのライブラリ。
コンパイル時、実行時で有用なメタ情報を取得できる。

```
struct info_printer {
  template <class IterInfo>
  void operator()(IterInfo) const
    { std::cout << IterInfo::type::full_name() << std::endl; }
};

template <typename T>
void print_info() {
  using namespace boost::mirror;
  typedef BOOST_MIRRORED_CLASS(T) meta_X;

  std::cout << meta_X::full_name() << std::endl; // std::string
  std::cout << meta_X::base_name() << std::endl; // string
  mp::for_each_ii< class_layout<meta_X> >(info_printer()); // メンバ列挙…
}

print_info<std::string>();
```

# STM

Software Transactional Memoryのライブラリ。
Lock/Unlock版のGCみたいなもの。

ロックして即処理、ではなくトランザクションによって管理するため、より高速で、ロックを行う他の処理とも合成できる。

```
boost::stm::tx::object<int> counter(0);

int increment() {
    BOOST_STM_TRANSACTION {
        return counter++;
    } BOOST_STM_END_TRANSACTION;
}
```

Boost C++ Libraries

# LA

新たな線形代数ライブラリ。
N次元のベクトル等に対して、operator|()による統一的なアクセス方法を提供する。
また、vector_traitsをユーザー定義型で特殊化することで、LAのベクトル、行列として簡単に扱うことができる。

```
struct float3 { float a[3]; };

namespace boost { namespace la {
  template <> struct vector_traits<float3> { ... };
}}
```

```
float3 v;
v|X = 0;
v|Y = 0;
v|Z = 7;
float vmag = magnitude(v);
float3 m = rotx_matrix<3>(3.14159f);
float3 vrot = m * v;
```

# UnderConstructionまとめ

- 夢のようなライブラリがたくさん作られてます
- ただ、レビューマネージャー不足でなかなかリリースできないのが現状
- 自分が得意な分野のライブラリがレビューキューにあったら、ぜひレビューマネージャーに立候補してください！

Boost C++ Libraries

# 全体まとめ

- Boostの日本語翻訳プロジェクトや稲葉さんの本などで騒がれていたのは1.32.0～1.34.0。
- 最新のBoostはさらなる進化を遂げており、Boostがあればたいていのことはできます。
- ぜひ仕事でも採用しましょう。
  自分で作るよりバグが少なく、多くのユーザーに使われてますよ。