

Boost.Intrusive × Boost.MultiIndex

k.inaba (稲葉一浩)
<http://www.kmonos.net/>

Boost.勉強会 Dec 12, 2009

今日お話したい内容

「僕は

データ構造

が大好きです」

Boostの すごい データ構造

その1

Boost.MultiIndex



日本語で言うと

Multi : 複数の

Index : 索引

お題：今日の発表リスト



```
#include <list>
```

```
struct Presen {  
    string twitterID; // ツイッターID  
    string hatenaID;  // はてなID  
    string title;      // 発表題目  
};  
std::list<Presen> timeTable;
```

お題：今日の発表リスト作ろう

```
#include <boost/assign.hpp>
```

```
timeTable += Presen(  
    “cpp_akira”,  
    “faith_and_brave”,  
    “Boostライブラリー週の旅”  
);
```

お題：今日の発表リスト作ろう

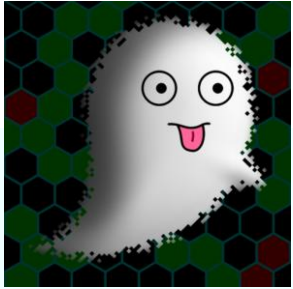
```
timeTable += Presen(  
    “kinaba”,  
    “cafelier”,  
    “Boost.MultiIntrusivedex”  
);
```

お題：今日の発表リスト作ろう

```
timeTable += Presen(  
    “melponn”,  
    “melpon”,  
    “Boost.Coroutine”  
);
```

・・・以下略・・・

作ったデータを使おう！



さんの発表タイトルって
なんだったっけ？

```
twid_to_title(  
    timeTable,  
    “DecimalBloat”  
) == “Boost.Preprocessor”
```

こうだ！

```
#include <boost/foreach.hpp>

string twid_to_title(tt, id)
{
    BOOST_FOREACH(Presen& p, tt)
        if( p.twitterID == id )
            return p.title;
    throw std::out_of_range(id);
}
```

こうだ！

```
#include <boost/foreach.hpp>

string twit_title(tt, id)
{
    BOOST_FOREACH(p, tt)
        if( p.tweet.ID == id )
            return p.title;
    throw std::runtime_error(id);
}
```

SAMPLE

Boost 勉強会 発表者

未来予測

○年後	西暦	魔法先生 人数	類似の数
1	2010	28	魔法先生
2	2011	56	AKB48
3	2012	112	株式会社
4	2013	224	東方project
5	2014	448	ポケモン
-	-	-	-

29	2038	7,516,192,768	世界人口
30	2039	15,032,385,536	
31	2040	30,064,771,072	

発表者

300億人に

スケールするには？

データ構造を変えれば

- `std::set` なら、検索が速い！

```
struct ByTwID
```

```
{ bool operator()(略) { 略 } };
```

```
// std::list<Presen>
```

```
std::set<Presen, ByTwID>  
    timeTable;
```

1億倍速い！

```
string twid_to_title(tt, id)
{
    auto it =
        tt.find(Presen(twid, "", ""));
    if( it != tt.end() )
        return it->title;
    throw std::out_of_range(twid);
}
```

1億倍速い！

```
string twid = ... (tt, id)
{
    auto it =
        tt.find(Prasen(twid, "", ""));
    if( it != Prasen() )
        return it + 1;
    throw std::runtime_error(twid);
}
```

SAMPLE

なんで不合格？ (1)

- ・ Twitter ID じゃなくて
はてな ID でも検索したい！

```
hatenaid_to_title(  
    timeTable,  
    “xyuyux”  
)  
    == “Boost.Asio”
```

なんで不合格？ (2)

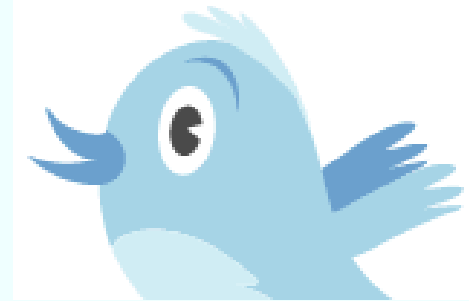
- ・ 発表の順番も知りたい！

```
BOOST_FOREACH(Presen& p, timTbl)  
    cout << p.title << endl;
```

```
// setだとIDの順に並んでしまう  
// - Boost.SmartPtr:shared_ptr+weak_ptr  
// - Boost.Preprocessor  
// - Boost.ライブラリー週の旅 ...
```

ここまでの要望をまとめる

Twitter ID で
高速検索できて



はてな ID で
高速検索できて

表に入れた順番も
覚えとけ！



そんな

Boost.MultiIndex

あなたに

こうだ！

```
typedef
multi_index_container<Presen,
    indexed_by<
        ordered_unique<
            member<Presen, string, &Presen::twitterID>
        >,
        ordered_unique<
            member<Presen, string, &Presen::hatenaID>
        >,
        sequenced<>
    >
> MyTimeTable;
```

入れる物は
Presen

.twitterID で
検索したい

.hatenaID で
検索したい

入れた順も
覚えといて！

$mi \langle Presen, index \langle Tw, Ht, seq \rangle \rangle$

```
// get<0>  
//     twitterIDで  
//     高速検索
```



```
timeTable.get<0>()  
    .find("wraith13");
```

$mi \langle Presen, index \langle Tw, Ht, seq \rangle \rangle$

```
timeTable.get<1>()  
    .find("Cryolite");
```



```
// get<1>  
// はてなIDで  
// 高速検索
```

$mi \langle Presen, index \langle Tw, Ht, seq \rangle \rangle$

// get<2> 入れた順

```
BOOST_FOREACH(  
    const Presen& p,  
    timeTable.get<2>()  
)  
    cout << p.title << endl;
```



よくある

質問

3つデータ構造
作るのと
何が違うの？

つまり、これとの違いは？

```
struct MyTimeTable {  
    set<Presen, ByTwID>          tt1;  
    set<Presen, ByHatenaID>      tt2;  
    list<Presen>                 tt3;  
    void add(const Presen& p)  
    { tt1.insert(p);  
      tt2.insert(p);  
      tt3.push_back(p); }  
};
```

つまり、これとの違いは？

```
struct MyTimeTable {  
    set<Present, MyTwID> tt1;  
    set<Present, MyHatenalID> tt2;  
    list<Present> tt3;  
    void add(const Present& p)  
    { tt1.insert(p);  
      tt2.insert(p);  
      tt3.push_back(p); }  
};
```

SAMPLE

インデックスの更新

「@kinaba です。用事入ったので発表キャンセル！」

```
// TwitterID検索用データから削除。OK  
tt1.erase(Presen("kinaba", "", ""));  
// HatenaID検索用データから削除。OK  
tt2.erase(...略...);  
// 発表順リストから削除... 遅い！  
tt3.erase(...略...);
```

31	2040	30,064,771,072
----	------	----------------

インデックスの更新

MultiIndex なら

```
tt.get<0>().erase("kinaba");
```

1 行 & 一瞬で
全インデックスから削除

用意されてるインデックス

	挿入	削除	機能
<code>ordered_unique</code> <code>ordered_non_unique</code>	$O(\log N)$	$O(1)$	<code>set</code> , <code>multiset</code> : 指定キーで検索
<code>hashed_unique</code> <code>hashed_non_unique</code>	$O(1)$	$O(1)$	<code>unordered_set</code> 等: 指定キーでハッシュ検索
<code>sequenced</code>	$O(1)$	$O(1)$	<code>list</code> : 入れた順を覚える
<code>random_access</code>	$O(1)$	$O(\text{後続要素数})$	<code>vector</code> : ランダムアクセス

MultiIndexの便利な所まとめ

- ・ 色々なインデックスを付けられる
- ・ `get<n>()` で普通のコンテナ扱い

```
tt.get<0>().erase("uskz");  
BOOST_FOREACH(p, tt.get<2>()) { ... }
```

- 整数 `n` でなく、タグも付けられます
- ・ 実はむしろ普通のコンテナより便利
 - `find(key)`, `modify(...)`

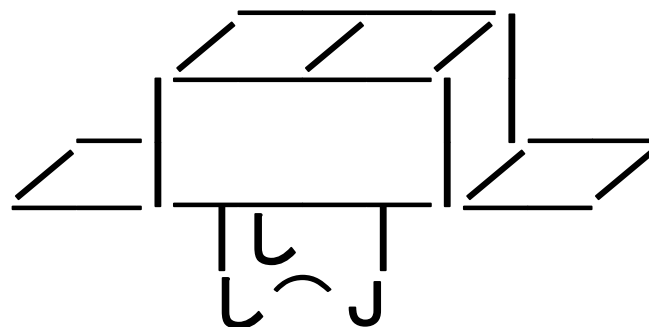
```
set_tt.find(Presen("uskz","", ""));  
mi_tt.find("uskz");
```


Boostの すごい データ構造

その2

Boost.Intrusive

こちらスネーク Boostへの
侵入に成功した



日本語で言うと

intrusive
= 侵入的

普通のデータ構造

- ・ 構造の管理用メンバは、外

```
struct Presen {  
    string twitterID;  
    string hatenaID;  
    string title;  
};
```

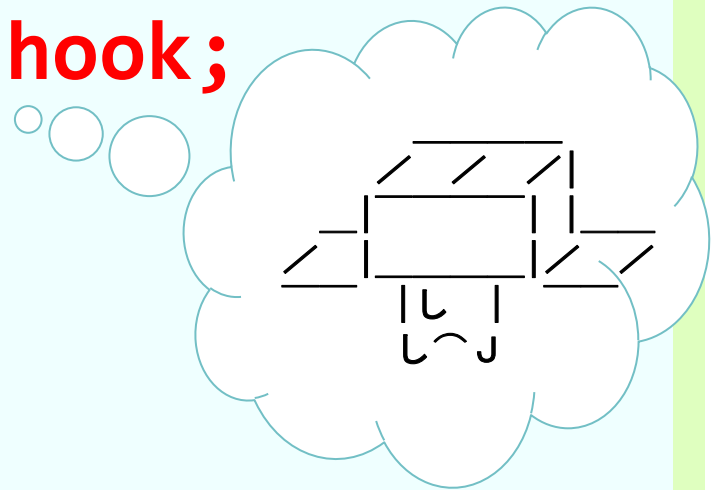
```
std::list<Presen> lst;
```

```
struct _List_node {  
    _List_node* prev;  
    _List_node* next;  
    Presen      data;  
};
```

侵入的データ構造

- ・ 管理用メンバが、侵入

```
struct Presen {  
    list_member_hook<> hook;  
    string twitterID;  
    string hatenaID;  
    string title;  
};  
intrusive::list<Presen,...> lst;
```



メリット&デメリット

・ デメリット

[Bad]

- データ定義がコンテナに依存しちゃう
- hook を自分で置く手間がかかる

・ メリット

[Good]

- コピー不可オブジェクトも入れられる
- 複数のコンテナに同時に入れられる
 - ・ 実は MultiIndex 的なことができる

メリット&デメリット

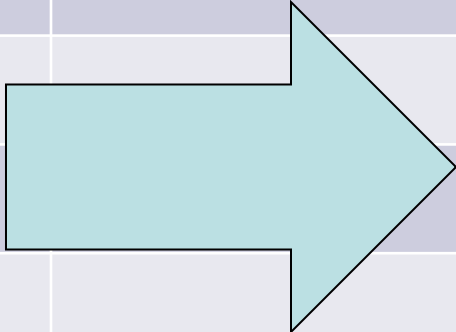
- ・ デメリット *[Bad]*
 - データ定義がコンテナに依存しちゃう
 - hook を自分で置く手間がかかる
 - ・ メリット *[Good]*
 - コピー不可オブジェクトも入れられる
 - 複数のコンテナに同時に入れられる
- 今日は細かい話はスキップ
- ・ 実は MultiIndex 的なことができる

データ構造マニア的 *Boost.Intrusive* の、特徴

異様に *set* の種類が多い

40/65

STL	MultiIndex	Intrusive
vector	random_access	
deque		
slist		slist
list	sequenced	list
set	ordered	set
unordered_set	hashed	unordered_set
		avl_set
		splay_set
		sg_set
		treap_set



マニア心をくすぐるset

- set 挿入:速い 検索:遅い
- avl_set 挿入:遅い 検索:速い
- sg_set 実行時↑バランスク替
- splay_set よく使う要素:速い
- treap_set 優先度つけられる

Boostの すごい データ構造

その3

まぜてみよう！

こちらスネークMultiIndexへの
侵入に成功した



すごいところ

- **MultiIndex** は
 - 入れるデータに手を加えずに
複数のインデックス張れて凄い
- **Intrusive** は
 - `sg_set` とか `splay_set` とか
色々あってすごい

あわせると・・・？

もっとすごい！

というわけで目標

- Intrusive を使って
MultiIndex 用インデックス

avl<>

splay<>

sg<>

treap<>

を実装します

インデックスの作り方調べ中...



Boost.MultiIndex Future work

将来やりたいなーと思ってる事

- User-defined indices

ユーザもインデックスを定義できるようにする

- The mechanisms by which Boost.MultiIndex orchestrates the operations of the indices held by a `multi_index_container` are simple enough to make them worth documenting so that the (bold) user can

もしかして：今はまだできない

インデックスの作り方調べ中...



Boost.MultiIndex Future work

将来やりたいなーと思ってる事

- User-defined indices

ユーザもインデックスを定義できるようにする

- The mechanisms by which Boost.MultiIndex orchestrates the operations of the indices held by a `multi_index_container` are simple enough to make them worth documenting so that the (bold) user can

もしかして

でも十分シンプル！ (ドキュメントないけど！)

というわけで

- ・この後の発表内容は
 - 私がソースから勝手に解釈した方法
 - 色々間違ってるかも
 - 将来的に色々変わるかも



この記事には『**独自研究**』に基づいた記述が含まれているおそれがあります。これを解消するためにある情報の根拠だけではなく、**信頼可能な解釈、評価、分析、総合の根拠**となる出典を示してください。

そもそも
複数のインデックス
の実体は？

mi < *T*, *index* < *Tw*, *Ht*, *Seq* >

・こんなノードクラス

```
class node_type {  
    node_type* twL, *twR;  
    redblack twC;  
    node_type* htL, *htR;  
    redblack htC;  
    node_type *prev;  
    node_type *next;  
    T value;  
}
```

正確には
継承で

```
class  
index_node_base  
{ T value; }
```

```
class {  
    seqnode* next;  
    seqnode* prev; }
```

```
class {  
    htnode *L, *R;  
    redblack C; }
```

```
class node_type {  
    node_type *L, *R;  
    redblack C; }
```

俺々ノード の 実装

intrusive::sg_set の場合

- ・ 親クラスを外から指定可にする

```
template<typename Super>
struct my_node : public Super
{
    sg_set_member_hook<> hook_;
};
```

あとは好きなように実装

コンストラクタ
呼ばれない
ので注意

次に インデックス本体 の实体

これも継承

```
class (順番保存用) : ...  
{ push_back, begin, end, ... }
```

↑ *protected*

```
class (はてなID検索用) : ...  
{ find, erase, begin, end, ... }
```

↑ *protected*

```
class (TwitterID検索用) : ...  
{ find, erase, begin, end, ... }
```

↑ *public*

```
class multi_index_container : ... {  
public: twIndex& get<0>() { return *this; }  
       htIndex& get<1>() { return *this; }  
       seqIndex& get<2>() { return *this; }  
}
```

```
class index_base{...}
```

↑ *protected*

俺々インデックス の 実装

intrusive::sg_set の場合

- ・ 親クラスを外から指定可にする

```
template<class Meta, class Tag>
struct my_index
    : protected Meta::type
{
    // ここで必須typedefを大量定義
    // ここで必須メソッドを実装する
    sg_set<my_node<...>> impl_;
};
```

あとは好きなように実装

次に コンテナ的メソッド の実装

例

pop_back

(最後の要素を消す)

```
class my_index : ... {  
    void pop_back() {  
        { my_node* p = 気合い;  
          final_erase_(p); }  
    void erase_(my_node* p)  
    { impl_.erase(気合(p));  
      super::erase_(p); } }
```

いろいろなインデックス

```
class index_base {  
    final_erase_(p){  
        ((mi*)this)  
        ->erase_(p);  
    } }
```

いろいろなインデックス

```
class multi_index_container : ...  
    { void erase_(node* p){super::erase_(p);} }
```

実装に使えるメソッド

- `bool final_empty_()`
- `size_t final_size_()`
- `size_t final_max_size_()`
- `pair<fnode*,bool> final_insert_(value)`
- `pair<fnode*,bool> final_insert_(value, fnode*)`
- `void final_erase_(fnode*)`
- `void final_delete_node_(fnode*)`
- `void final_delete_all_nodes_()`
- `void final_clear_()`
- `void final_swap_(final&)`
- `bool final_replace_(value, fnode*)`
- `bool final_modify_(MODIFIER mod, fnode*)`
- `bool final_modify_(MODIFIER mod, ROLLBACK back, fnode*)`

実装しないといけないメソッド

- `void copy_(const self&, const copy_map_type&)`
- `node* insert_(value, node*)`
- `node* insert_(value, node* position, node*)`
- `void erase_(node*)`
- `void delete_node_(node*)`
- `void delete_all_nodes_()`
- `void clear_()`
- `void swap_()`
- `bool replace_(value, node*)`
- `bool modify_(node*)`
- `bool modify_rollback_(node*)`

最後に

IndexSpecifier

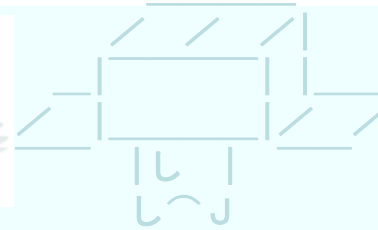
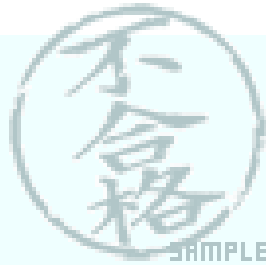
IndexSpecifierとは？

```
multi_index_container<Presen,  
    indexed_by<ここに並べるもの>  
>
```

```
template<typename Tag=tag<>>  
struct use_my_index {  
    template<typename Super>  
        struct node_class  
            { typedef my_node<Super> type; };  
    template<typename Meta>  
        struct index_class  
            { typedef my_index<Meta,Tag> type; }  
};
```

完成！

Thank You for Listening!



<http://github.com/kinaba/mint>

