Boost+ライブラリー周の旅

高橋晶 (Akira Takahashi)

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

Boost.勉強会 2009/12/12(Sat)

はじめに

このセッションは

- Boostに興味があるけど触ったことがない
- バージョンアップについていけなくなった
- Boostの全容を知りたい

といった方のために、Boost 1.40.0時点での
なるべく全てのライブラリの概要を知ってもらう
ためのものです

Boost C++ Libraries

本日紹介するライブラリ

01.Accumulators
02.Any
03.Array
04.Asio
05.Assign
06.Bimap
07.Bind
08.Circular Buffer
09.Compressed Pair
10.Concept Check
11.Conversion
12.CRC
13.Date Time
14.Dynamic Bitset
15.Enable If
16.Exception
17.Filesystem
18.Flyweight
19.Foreach
20.Format

21.Function
22.Function Types
23.Fusion
24.GIL
25.Graph
26.Interprocess
27.Interval
28.Intrusive
29.IO State Server
30.Iostreams
31.Iterators
32.Lambda
33.Math
34.Member Function
35.MPL
36.Multi Array
37.Multi Index
38.Numeric Conversion
39.Operators
40.Optional

41.Parameter
42.Pointer Container
43.Pool
44.Preprocessor
45.Property Map
46.Proto
47.Python
48.Random
49.Range
50.Ref
51.Scope Exit
52.Serialization
53.Signals2
54.Smart Pointers
55.Spirit
56.Statechart
57.Static Assert
58.String Algo
59.Swap
60.System

61.Test
62.Thread
63.Timer
64.Tokenizer
65.Tribool
66.Tuple
67.Typeof
68.uBLAS
69.Units
70.Unordered
71.Utility
72.Variant
73.Wave
74.Xpressive

Boost C++ Libraries

はじめる前に

1分でわかるテンプレートメタプログラミング

Boost C++ Libraries テンプレートメタプログラミングとは

テンプレートのインスタンス化を利用して

あらゆるコンパイル時計算を行うパラダイム。

プログラミングの対象はプログラム自身のメタな情報

Boost C++ Libraries

メタ関数

テンプレートパラメータ　　：関数のパラメータ
クラス中のtypedef　　：戻り値
と見なしたクラスを**メタ関数**という。

条件分岐　：テンプレートの特殊化
ループ　　：メタ関数の再帰呼び出し
で表現できる。

Boost C++ Libraries

メタ関数の例

T型を受け取り、N個の*(ポインタ)付加した 型を返すメタ関数

```
template <class T, int N> // パラメータ
struct add_ptrs {
  // 再帰
  typedef typename add_ptrs<T*, N-1>::type type;
};

template <class T>
struct add_ptrs<T, 0> { // 条件分岐
  typedef T type; // 戻り値
};

typedef add_ptrs<int, 3>::type type; // 呼び出し
// type == int***
```

では今度こそ

「Boostライブラリ 一周の旅」

はじめます！

Boost C++ Libraries

Accumulators

拡張可能な統計計算フレームワーク

```
using namespace boost;

accumulator_set<int, features<tag::min, tag::sum> > acc;

acc(1);
acc(2);
acc(3);

cout << "Min: " << min(acc) << endl; // 1
cout << "Sum: " << sum(acc) << endl; // 6
```

Any

あらゆる型を保持できる動的型

```
list<boost::any> ls;
ls.push_back(1);              // int
ls.push_back(string("abc")); // string
ls.push_back(3.14);           // double

while (!ls.empty()) {
  boost::any& a = ls.front();

  if (a.type() == typeid(int)) { int i = boost::any_cast<int>(a); }
  if (a.type() == typeid(string)) ...
  if (a.type() == typeid(double)) ...

  ls.pop_front();
}
```

Array

配列(コンテナのインタフェースが使える)

```
boost::array<int, 3> ar = {1, 2, 3};

for (size_t i = 0; i < ar.size(); ++i)
  cout << ar[i] << endl;

for_each(ar.begin(), ar.end(), f);
```

Asio

非同期ネットワークライブラリ

```
using namespace boost::asio;

void connection(int port)
{
  io_service io;
  tcp::acceptor acc(io, tcp::endpoint(tcp::v4(), port));

  for (;;) {
    tcp::iostream s;
    acc.accept(*s.rdbuf());

    string line;
    while (getline(s, line)) {
      cout << line << endl;
    }
  }
}
```

Assign

コンテナの簡易構築

```
using namespace boost::assign;

vector<int> v;
v += 3, 1, 4;

list<int> ls = list_of(3)(1)(4);

map<string, int> m;
insert(m)("Akira", 24)("Millia", 16)("Johnny", 38);
```

Boost C++ Libraries

Bimap

双方向map

bimap<X, Y>は、std::map<X, Y>とstd::map<Y, X>両方の用途

```
typedef boost::bimaps::bimap<int, string> bm_type;

bm_type m;

m.left.insert(bm_type::left_value_type(3, "Akira"));
m.right.insert(bm_type::right_value_type("Millia", 1));

cout << m.left.at(3)          << endl; // Akira
cout << m.right.at("Millia") << endl; // 1

cout << m.right.at(“Akira”)  << endl; // 3
cout << m.left.at(1)          << endl; // Millia
```

Bind

部分評価

```
void foo(int x, int y) {} // 3 : 3と4が渡される

template <class F>
void bar(F f)
{
  f(4); // 2 : 残りの引数を渡す
}

bar(boost::bind(foo, 3, _1)); // 1 : 2引数のうち、1つだけ渡す
```

Boost C++ Libraries

Circular Buffer

循環バッファ
バッファがいっぱいになったら上書きしていく

```
boost::circular_buffer<int> buff(3);

buff.push_back(1); buff.push_back(2); buff.push_back(3);

int a = buff[0]; // a : 1
int b = buff[1]; // b : 2
int c = buff[2]; // c : 3

buff.push_back(4); buff.push_back(5);

a = buff[0]; // a : 3
b = buff[1]; // b : 4
c = buff[2]; // c : 5
```

Boost C++ Libraries

Compressed Pair

テンプレート引数のどちらかが空クラスだった場合に
最適化されやすいpair

```
struct hoge {}; // empty class

boost::compressed_pair<hoge, int> p(hoge(), 1);

hoge& h = p.first();
int&  i = p.second();
```

Concept Check 1/3

テンプレートパラメータの制約

(C++0xでお亡くなりになられたアレのライブラリ版)

```
template <class Iterator>
void my_sort(Iterator first, Iterator last)
{
  BOOST_CONCEPT_ASSERT((boost::RandomAccessIterator<Iterator>));
  std::sort(first, last);
}

list<int> ls;
my_sort(ls.begin(), ls.end());
```

Boost C++ Libraries — Concept Check 2/3

Concept Checkを使わない場合のエラーメッセージ(VC9)

```
error C2784:
'reverse_iterator<_RanIt>::difference_type
  std::operator -(const std::reverse_iterator<_RanIt> &,const
                        std::reverse_iterator<_RanIt2> &)'
: テンプレート 引数を 'const std::reverse_iterator<_RanIt> &' に対して
'std::list<_Ty>::_Iterator<_Secure_validation>' から減少できませんでした

'std::operator -' の宣言を確認してください。
...
```

全然わからない！

Concept Check 3/3

Concept Checkを使った場合のエラーメッセージ(VC9)

error C2676:
二項演算子 '+=' : 'std::list<_Ty>::_Iterator<_Secure_validation>' は、
この演算子または定義済の演算子に適切な型への変換の定義を行いません。

クラス テンプレート のメンバ関数
'boost::RandomAccessIterator<TT>::~RandomAccessIterator(void)' の
コンパイル中
...

かなりよくなった

Conversion

型変換ライブラリ

```
// lexical_cast : 数値と文字列の相互変換
int         n = boost::lexical_cast<int>(“123”);
std::string s = boost::lexical_cast<std::string>(123);

Base* b;

// polymorphic_downcast : アサート + static_cast
Derived* d = boost::polymorphic_downcast<Derived*>(b);

// polymorphic_cast : 失敗時は例外を投げるdynamic_cast
Derived* d = boost::polymorphic_cast<Derived*>(b);
```

CRC

CRC計算

```
// "123456789"のASCIIコード
unsigned char const data[] =
    { 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37, 0x38, 0x39 };
std::size_t const data_len = sizeof(data) / sizeof(data[0]);

boost::uint16_t const expected = 0x29B1;

// CRC-CCITT
boost::crc_basic<16> crc_ccitt1(0x1021, 0xFFFF, 0, false, false);
crc_ccitt1.process_bytes(data, data_len);
assert(crc_ccitt1.checksum() == expected);
```

Date Time

日付・時間ライブラリ

```
using namespace boost::gregorian;
using namespace boost::posix_time;

ptime now = second_clock::local_time();

// 日付計算
date today    = now.date();
date tomorrow = today + date_duration(1);

// 時間計算
ptime t = now + minutes(3);
```

Boost C++ Libraries

Dynamic Bitset

大きさを動的に変えられるbitset

```
boost::dynamic_bitset<> bs(10);

// 偶数番目のビットを立てる
for (size_t i = 0; i < bs.size(); ++i) {
  if (i % 2 == 0)
    bs[i] = 1; // 添字アクセス
}

cout << bs << endl; // 0101010101
```

Enable If

型特性によるオーバーロード

```
template <class T>
void f(T x, typename enable_if<is_integral<T> >::type* = 0)
  { cout << "整数型" << endl; }

template <class T>
void f(T x, typename disable_if<is_integral<T> >::type* = 0)
  { cout << "整数型以外" << endl; }

int i; char c; double d;

f(i); // int    : 整数型
f(c); // char   : 整数型
f(d); // double : 整数型以外
```

Exception

catchする度にエラー情報を付加する

```
class MyException : public boost::exception, public std::exception {};
typedef boost::error_info<struct tag_errmsg, string> error_message;

void g() { BOOST_THROW_EXCEPTION(MyException()); }

void f() {
  try { g(); }
  catch (MyException& e) {
    e << error_message("何か悪いことをした"); // エラー情報を付加して
    throw; // 再スロー
  }
}

try { f(); }
catch (MyException& e) { // 階層的に情報が付加された例外を受け取る
  cout << boost::diagnostic_information(e) << endl; // 表示
}
```

Filesystem

パス、ファイル、ディレクトリ操作

```
using namespace boost::filesystem;

remove_all(“my_dir”);            // ディレクトリ内のファイル削除

create_directory(“my_dir”);      // ディレクトリ作成

ofstream file(“my_dir/a.txt”); // ファイル書き込み
file << “test¥n";
file.close();

if (!exists(“my_dir/a.txt”))    // ファイルの存在チェック
  std::cout << “ファイルがない¥n";
```

Flyweight

リソースの共有

```
using boost::flyweights::flyweight;

flyweight<std::string> f1("abc");
flyweight<std::string> f2("abc");

// f1とf2は同じオブジェクトを指している
assert(&f1.get() == &f2.get());
```

Boost C++ Libraries

Foreach

foreach文。コンテナ/配列を順番に処理する

```
vector<string> v;
v.push_back("abc");
v.push_back("123");
v.push_back("xyz");

BOOST_FOREACH (const string& s, v) {
  cout << s << endl;
}
```

```
abc
123
xyz
```

Format

文字列のフォーマット

```
// sprintf風のフォーマット
string s1 = (boost::format("this year is %d.") % 2009).str();
cout << s1 << endl; // this year is 2009.

// プレースホルダーによるフォーマット
string s2 = (boost::format("next year is %1%.") % 2010).str();
cout << s2 << endl; // next year is 2010
```

Boost C++ Libraries

Function

汎用関数オブジェクト

テンプレート引数は関数の型ではなく関数の形(戻り値の型とパラメータの型)

```
int func(double) { return 1; }

struct functor {
  typedef int result_type;
  int operator()(double) const { return 2; }
};

// 関数ポインタ
boost::function<int(double)> f1 = func;
int r1 = f1(3.14);

// 関数オブジェクト
boost::function<int(double)> f2 = functor();
int r2 = f2(3.14);
```

Boost C++ Libraries

Function Types

関数の型情報を取得するメタ関数

```
using namespace boost::function_types;

// 型が関数ポインタかどうか判別
bool b = is_function_pointer<bool(*)(int)>::value; // == true

// 関数(関数ポインタ or 関数オブジェクト)の戻り値の型を取得
typedef result_type<bool(&)(int)>::type result_type; // is bool
```

Fusion

様々なデータ構造を持つ

コンパイル時＆実行時タプルライブラリ

```
struct disp {
  template <class T>
  void operator()(T x) const { cout << x << endl; }
};

using namespace boost::fusion;

// コンパイル時のタプル(型リスト)操作：一番後ろの型を取り除く
typedef vector<int, double, string, string>          typelist;
typedef result_of::pop_back<typelist>::type          unique_typelist;
typedef result_of::as_vector<unique_typelist>::type vector_type;

// 実行時のタプル操作：タプルの全要素を出力
vector_type v(1, 3.14, "abc");
for_each(v, disp());
```

GIL

画像処理

```
using namespace boost::gil;

rgb8_image_t img;
jpeg_read_image("a.jpg", img);

// 100x100にリサイズ
rgb8_image_t square (100, 100);
resize_view(const_view(img),
            view(square),
            bilinear_sampler());
jpeg_write_view("out-resize.jpg", const_view(square));
```

Graph

グラフ構造

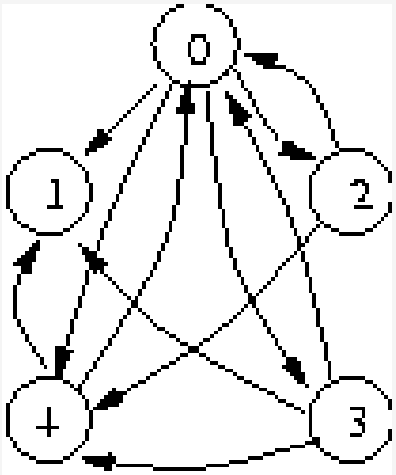
```
typedef adjacency_list<vecS, vecS, bidirectionalS> Graph;

// 頂点のため便宜上のラベルを作る
enum { A, B, C, D, E, N };
const int num_vertices = N;

// グラフの辺を書き出す
typedef std::pair<int, int> Edge;
Edge edge_array[] =
  { Edge(A,B), Edge(A,D), Edge(C,A), Edge(D,C),
    Edge(C,E), Edge(B,D), Edge(D,E) };
const int num_edges = sizeof(edge_array)/sizeof(edge_array[0]);

Graph g(num_vertices); // グラフオブジェクトを宣言

// グラフオブジェクトに辺を追加
for (int i = 0; i < num_edges; ++i)
  add_edge(edge_array[i].first, edge_array[i].second, g);
```



Interprocess

プロセス間共有メモリ

```
int main(int argc, char* argv[])
{
  using namespace boost::interprocess;
  typedef pair<double, int> MyType;

  if (argc == 1) {
    managed_shared_memory shm(create_only, "MySharedMemory", 128);

    // MyTypeのオブジェクトを作成して初期化
    MyType* a = shm.construct<MyType>("MyType instance")(0.0, 0);
  } else {
    managed_shared_memory shm(open_only, "MySharedMemory");

    pair<MyType*, size_t> res = shm.find<MyType>("MyType instance");
    shm.destroy<MyType>("MyType instance");
  }
}
```

Interval

区間計算

```
using namespace boost::numeric;

// 区間内かどうかのチェック
interval<double> range(1.0, 5.0);
assert(in(3.0, range));

// 区間同士の計算
interval<double> x(2, 3);
interval<double> y(1, 4);
interval<double> z = x + y;
cout << z << endl; // [3, 7]
```

Intrusive

侵入コンテナ

オブジェクトのコピーではなくオブジェクト自身を格納する

```
using namespace boost::intrusive;

class Window : public list_base_hook<> {
public:
  typedef list<Window> window_list;
  static window_list windows;

  Window()          {windows.push_back(*this);}
  virtual ~Window(){windows.erase(window_list::s_iterator_to(*this));}
  virtual void Paint() = 0;
};
Window::window_list Window::windows;

void paint_all_windows() {
   for_each(Window::windows, boost::mem_fn(&Window::Paint));
}
```

Boost C++ Libraries

IO State Server

IO Streamの状態管理

```
void hex_out(std::ostream& os, int x)
{
  boost::io::ios_flags_saver ifs(os);
  os << std::hex << x << endl;
} // ここでstreamの状態が戻る

int x = 20;
cout << x << endl; // 20 : 10進(状態変更前)
hex_out(cout, x);  // 14 : 16進(状態変更)
cout << x << endl; // 20 : 10進(状態が戻ってる)
```

Boost C++ Libraries

Iostreams

拡張IO Streamライブラリ。

streamクラスを簡単に作るLibrary for Librariesであり、

パイプ演算子によるフィルタ設定, etc...

```
namespace io = boost::iostreams;
struct upper_filter : io::stdio_filter {
  void do_filter() {
    int c;
    while ((c = std::cin.get()) != EOF)
      std::cout.put(std::toupper((unsigned char)c));
  }
};
BOOST_IOSTREAMS_PIPABLE(upper_filter, 0)

// 大文字に変換して、gzip圧縮して、ファイルに出力
io::filtering_ostream out(upper_filter()          |
                          io::gzip_compressor()   |
                          io::file_sink("a.txt"));
out << "aiueo" << std::endl;
```

Boost C++ Libraries

Iterators

イテレータを簡単に作るためのライブラリ

```
class count_iterator : public boost::iterator_facade<
        count_iterator, const int, boost::forward_traversal_tag> {
public:
count_iterator(int x) : x_(x) {}
private:
  friend class boost::iterator_core_access;

  void        increment() { ++x_; }
  const int& dereference() const { return x_; }
  bool        equal(const count_iterator& other) const
                { return x_ == other.x_; }

  int x_;
};

copy(count_iterator(0), count_iterator(5),
       ostream_iterator<int>(cout, “")); // 01234
```

Boost C++ Libraries

Lambda

ラムダ式。その場で関数オブジェクトを作成する

```
using namespace boost::lambda;

vector<int> v;
v.push_back(1); v.push_back(2); v.push_back(3);

for_each(v.begin(), v.end(), cout << _1 << ' '); // 1 2 3
```

cout << _1 << ‘ ’;
でこんな感じの関数オブジェクトができると思ってもらえれば。

```
struct F {
  template <class T>
  ostream& operator()(const T& x) const
    { return cout << x << ‘ ’; }
};
```

Math

数学の特殊関数とか

```
using namespace boost::math;

cout << factorial<double>(3) << endl; // 階乗        : 6
cout << round(3.14)          << endl; // 四捨五入    : 3
cout << gcd(6, 15)           << endl; // 最大公約数 : 3
```

Boost C++ Libraries

Member Function

std::mem_funとstd::mem_fun_refを一般化したもの

```
struct button {
  explicit button(const point p);
  void draw() const;
};

vector<button> v;
v.push_back(button( 10,  10));
v.push_back(button( 10,  30));
v.push_back(button(200, 180));

// 全てのbuttonのdrawメンバ関数を呼ぶ
for_each(v.begin(), v.end(), boost::mem_fn(&button::draw));
```

Boost C++ Libraries

MPL

テンプレートメタプログラミングのライブラリ

```
template <class T>
struct add_ptr { typedef T* type; };

using namespace boost::mpl;

typedef vector<int, char>                    vec1; //{int,char}
typedef push_back<vec1, double>::type        vec2; //{int,char,double}
typedef reverse<vec2>::type                  vec3; //{double,char,int}
typedef transform<vec3, add_ptr<_> >::type vec4; //{double*,char*,int*}

typedef vector<double*, char*, int*> result;
BOOST_MPL_ASSERT(( equal<vec4, result> )); // OK
```

Boost C++ Libraries

Multi Array

多次元配列

```
typedef boost::multi_array<int, 3> Array;
Array ar(boost::extents[3][4][2]);

int value = 0;
for (size_t i = 0; i < ar.size(); ++i)
  for (size_t j = 0; j < ar[i].size(); ++j)
    for (size_t k = 0; k < ar[i][j].size(); ++k)
      ar[i][j][k] = value++;
```

Multi Index

複数のソート順、アクセス順序を持たせることのできるコンテナ

```
using namespace boost::multi_index;
typedef multi_index_container<
  std::string,
  indexed_by<
    sequenced<>,
    ordered_non_unique<identity<std::string> >
  >
> container;

container words;
words.push_back("C++");
words.push_back("Action Script");
words.push_back("Basic");

copy(words, ostream_iterator<string>(cout, "¥n")); // #1 入れた順

const container::nth_index<1>::type& c = words.get<1>();
copy(c, ostream_iterator<string>(cout, "¥n")); // #2 辞書順
```

#1 入れた順(sequenced)
C++
Action Script
Basic

#2 辞書順(ordered)
Action Script
Basic
C++

Numeric Conversion

数値型の型変換

```
typedef boost::numeric::converter<int, double> DoubleToInt;

try {
  int x = DoubleToInt::convert(2.0);
  assert(x == 2);

  double m = boost::numeric::bounds<double>::highest();
  int y = DoubleToInt::convert(m);
                          // デフォルトではpositive_overflowを投げる
}
catch (boost::numeric::positive_overflow& ex) {
  cout << ex.what() << endl;
}
```

Operators

関連する演算子の自動生成

```
class person : boost::less_than_comparable<person> {
  int id_;
public:
  explicit person(int id) : id_(id) {}

  // operator<を定義すれば、>, <=, >=が自動生成される
  friend bool operator<(const person& lhs, const person& rhs)
    { return lhs.id_ < rhs.id_; }
};

person a(1);
person b(2);

bool c = a < b;
bool d = a > b;
bool e = a <= b;
bool f = a >= b;
```

Optional

有効な値と、無効な値

エラー値が-1だったり空文字列だったりするので統一する

```
boost::optional<int> find(const vector<int>& v, int x)
{
  vector<int>::const_iterator it = find(v.begin(), v.end(), x);
  if (it != v.end())
    return x;          // 見つかったら有効な値を返す
  return boost::none; // 見つからなかったら無効な値を返す
}

vector<int> v;
v.push_back(1); v.push_back(2); v.push_back(3);

boost::optional<int> p = find(v, 1);
if (p)
  cout << "found : " << p.get() << endl; // found : 1
else
  cout << "not found" << endl;
```

Parameter

名前付き引数

```
BOOST_PARAMETER_NAME(name)
BOOST_PARAMETER_NAME(age)

template <class Args>
void print(const Args& args)
{
  cout << args[_name] << ", " << args[_age] << endl;
}

print((_name="Akira", _age=24));
print((_age=24, _name="Akira"));
```

```
Akira, 24
Akira, 24
```

Pointer Container

ヒープオブジェクトを格納するためのコンテナ

スマートポインタのコンテナよりもコストが低い

```
struct drawable {
  virtual void draw() const = 0;
};
struct rectangle : drawable { void draw() const {} };
struct circle    : drawable { void draw() const {} };


boost::ptr_vector<drawable> v;

v.push_back(new rectangle());
v.push_back(new circle());

v.front().draw();
```

Boost C++ Libraries

Pool

メモリプール

```
struct X {
  ...
};

void f()
{
  boost::object_pool<X> pool;

  for (int i = 0; i < 1000; i++) {
    X* x = pool.malloc();

    ...xを使って何かする...
  }
} // ここでpoolのメモリが解放される
```

Boost C++ Libraries

Preprocessor

プリプロセッサメタプログラミングのライブラリ

コードの自動生成とかに使う(可変引数とか)

```
#define MAX 3
#define NTH(z, n, data) data ## n

int add(BOOST_PP_ENUM_PARAMS(MAX, int x))
{
  return BOOST_PP_REPEAT(MAX, NTH, + x);
}

assert(add(1, 2, 3) == 6);
```

```
int add( int x0 , int x1 , int x2)
{
  return + x0 + x1 + x2;
}

assert(add(1, 2, 3) == 6)
```

Property Map

インターフェースのマッピング

iterator_traitsの拡張版みたいなもの

```
template <class Assoc>
void foo(Assoc& m)
{
  typedef typename boost::property_traits<Assoc>::value_type type;

  type& value = get(m, "Johnny");
  value = 38;

  m["Akira"] = 24;
}

map<string, int> m;
boost::associative_property_map<map<string, int> > pm(m);

m.insert(make_pair("Millia", 16));
foo(pm);
```

Boost C++ Libraries

Proto

Expression Templateのライブラリを作るためのライブラリ

```
namespace proto = boost::proto;

proto::terminal<std::ostream&>::type cout_ = { std::cout };

template <class Expr>
void evaluate(const Expr& expr)
{
  proto::default_context ctx;
  proto::eval(expr, ctx);
}

evaluate(cout_ << "hello" << ',' << " world");
```

Boost C++ Libraries

Python

C++からPython、PythonからC++を使うためのライブラリ

1.C++の関数を用意する

```
const char* greet() { return "hello, world“; }
```

2.C++の関数をPython用にエクスポートする(DLLが作成される)

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

BOOST_PYTHON_MODULE(hello_ext)
{
    using namespace boost::python;
    def("greet", greet);
}
```

3.PythonでDLLをインポートしてC++関数を呼び出す

```
>>> import hello_ext
>>> print hello.greet()
hello, world
```

Boost C++ Libraries

Random

疑似乱数生成ライブラリ

疑似乱数生成器と分布の方法を分けて指定できるのが特徴

```
// 疑似乱数生成器：メルセンヌツイスター法
// 分布方法：一様整数分布(1〜10)

mt19937        gen(static_cast<unsigned long>(time(0)));
uniform_int<> dst(1, 10);
variate_generator<mt19937&, uniform_int<> > rand(gen, dst);

for (int i = 0; i < 10; ++i)
  cout << rand() << endl;
```

Boost C++ Libraries

Range

範囲に対する操作のためのユーティリティ

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


std::vector<int> v;
int ar[3];

std::vector<int>::iterator it = find(v,  3); // コンテナ
int*                       p  = find(ar, 3); // 配列
```

Ref

参照ラッパ

関数テンプレートのパラメータがT xになってるときにint valueを渡すとTがint&ではなくintに推論されてしまうのでboost::refで明示的に参照にする。boost::bindと組み合わせて使うことが多い。

```
void f(int& x)
{
  x = 3;
}

int x = 1;

boost::bind(f, x)();
cout << x << endl; // 1 : 変わってない

boost::bind(f, boost::ref(x))();
cout << x << endl; // 3 : OK
```

Boost C++ Libraries

Scope Exit

関数を抜けるときに実行されるブロックを定義する

(&variable)で変数を参照でキャプチャ

(variable)で変数をコピーでキャプチャ

```
class window {
  button back_button_;
  button next_button_;
public:
  void mouse_up()
  {
    BOOST_SCOPE_EXIT((&back_button_)(&next_button_)) {
      back_button_.up();
      next_button_.up();
    } BOOST_SCOPE_EXIT_END;

    …ボタンが押されたときの処理とか…

  } // ここで各ボタンのup()が呼ばれる
};
```

Boost C++ Libraries — Serialization

クラス情報のシリアライズ／デシリアライズ

```
class Person {
  int    age;
  string name;

  friend class boost::serialization::access;
  template <class Archive>
  void serialize(Archive& archive, unsigned int version)
  {
      archive & boost::serialization::make_nvp("Age", age);
      archive & boost::serialization::make_nvp("Name", name);
  }
};
std::ofstream ofs("C:/person.xml");
boost::archive::xml_oarchive oarchive(ofs); // XMLシリアライズ

Person person;
oarchive << boost::serialization::make_nvp("Root", person);
```

Boost C++ Libraries

Signals2

シグナル・スロット

```
struct back {
  void operator()() const { std::cout << "back" << std::endl; }
};

struct rollback {
  void operator()() const { std::cout << "rollback" << std::endl; }
};


boost::signals2::signal<void()> event;

event.connect(back());
event.connect(rollback());

event(); // back::operator(), rollback::operator()が呼ばれる
```

Boost C++ Libraries

Smart Pointers

スマートポインタライブラリ

```
void foo()
{
 // スコープ抜けたら解放されるスマートポインタ(コピー不可)
  boost::scoped_ptr<hoge*> h(new hoge());

  h->bar();

} // ここでdeleteされる
```

Spirit

構文解析

```
// "(1.0, 2.0)"という文字列をパースしてdoubleの組に入れる
string input("(1.0, 2.0)");

pair<double, double> p;
qi::parse(input.begin(), input.end(),
  '(' >> qi::double_ >> ", " >> qi::double_ >> ')',
  p);
```

Boost C++ Libraries

Statechart

状態マシン

```
struct StartStopEvent : event<StartStopEvent> {};
struct ResetEvent     : event<ResetEvent> {};

struct Active;  struct Stopped;
struct StopWatch : state_machine<StopWatch, Active> {};

struct Active : simple_state<Active, StopWatch, Stopped>
  { typedef transition<ResetEvent, Active> reactions; };

struct Running : simple_state<Running, Active>
  { typedef transition<StartStopEvent, Stopped> reactions; };

struct Stopped : simple_state<Stopped, Active>
  { typedef transition<StartStopEvent, Running> reactions; };

StopWatch watch;
watch.initiate();
watch.process_event(StartStopEvent()); // stop -> run
watch.process_event(StartStopEvent()); // run  -> stop
watch.process_event(StartStopEvent()); // stop -> run
watch.process_event(ResetEvent());     // run  -> stop
```

Boost C++ Libraries

Static Assert

コンパイル時アサート

Type Traitsと組み合わせてテンプレートの要件にしたり

コンパイル時計算の結果が正しいか検証するのに使う

```
template <class Integral>
bool is_even(Integral value)
{
  // 整数型じゃなかったらコンパイルエラー
  BOOST_STATIC_ASSERT(is_integral<Integral>::value);
  return value % 2 == 0;
}

is_even(3);    // OK
is_even(3.14); // エラー！
```

Boost C++ Libraries

String Algo

文字列のアルゴリズム

```
string str = "hello world ";

boost::to_upper(str);   // “HELLO WORLD ” : 大文字に変換
boost::trim_right(str); // “HELLO WORLD”  : 右の空白を除去

// 拡張子の判定
bool is_executable(const std::string& filename)
{
  return boost::iends_with(filename, ".exe");
}
```

Swap

std::swapの強化版

```
struct hoge {
  void swap(hoge&) {}
};

// 組み込み配列をswap可能
int a1[3];
int a2[3];
boost::swap(a1, a2);

// 専門特化したswapがあればそっちを使う。この場合はhoge::swap
hoge h1;
hoge h2;
boost::swap(h1, h2);
```

Boost C++ Libraries

System

各OSのエラーコードをラップして汎用化

```
namespace sys = boost::system;

try {
  // OS固有のAPIでエラーが発生したら
  WSADATA wsa_data;
  int result = WSAStartup(MAKEWORD(2, 0), &wsa_data);

  // error_codeでOS固有のエラー情報を取得してsystem_error例外を投げる
  if (result != 0) {
    throw sys::system_error(
            sys::error_code(result,
                            sys::error_category::get_system_category()),
            "winsock");
  }
}
catch (sys::system_error& e) {
  cout << e.code() << "," << e.what() << endl;
  throw;
}
```

Test

テストライブラリ

```
using namespace boost::unit_test_framework;

void size_test()
{
  std::vector<int> v;
  const int size = v.size();

  v.push_back(3);
  BOOST_CHECK_EQUAL(v.size(), size + 1);

  v.pop_back();
  BOOST_CHECK_EQUAL(v.size(), size);
}

test_suite* init_unit_test_suite(int argc, char* argv[])
{
  test_suite* test = BOOST_TEST_SUITE("test");
  test->add(BOOST_TEST_CASE(&size_test));
  return test;
}
```

```
Running 1 test case...

*** No errors detected
```

Boost C++ Libraries

Thread

スレッド

```
void hello()
{
  cout << "Hello Concurrent World" << endl;
}

int main()
{
  boost::thread t(hello);
  t.join();
}
```

Boost C++ Libraries

Timer

簡単な時間計測

```
boost::timer t;

// 時間のかかる処理...

cout << "処理時間 : " << t.elapsed() << "秒" << endl;
```

処理時間 : 3.109秒

Boost C++ Libraries

Tokenizer

トークン分割

```
void disp(const string& s) { cout << s << endl; }

string s = "This is a pen";
boost::tokenizer<> tok(s);

for_each(tok.begin(), tok.end(), disp);
```

```
This
is
a
pen
```

Boost C++ Libraries

Tribool

3値bool

```
using namespace boost::logic;

tribool a = true;  // 真
a = false;         // 偽
a = indeterminate; // 不定
```

Tuple

タプル

std::pairは2つの値の組だが、tupleは3つ以上の値も可能

```
using namespace boost::tuples;

tuple<int, string, double> get_info() // 多値を返す関数
{
  return make_tuple(5, "Hello", 3.14);
}


tuple<int, string, double> t = get_info();

int n;
double d;
tie(n, ignore, d) = get_info(); // 一部の値を取り出す(2番目はいらない)
```

Boost C++ Libraries

Typeof

型推論

C++0xのautoとdecltypeをエミュレーション

```
int a;

// 式の適用結果の型 : int b = a + 2;
BOOST_TYPEOF(a + 2) b = a + 2;

// 右辺の式から左辺の型を推論 : int c = b + 2;
BOOST_AUTO(c, b + 2);
```

C++0xではこうなる

```
int a;

// 式の適用結果の型 : int b = a + 2;
decltype(a + 2) b = a + 2;

// 右辺の式から左辺の型を推論 : int c = b + 2;
auto c = b + 2);
```

Boost C++ Libraries

uBLAS

ベクトルや行列といった線形代数のライブラリ

Expression Templateで組まれているため高速なのが特徴

```
using namespace boost::numeric::ublas;

vector<double> a(3);
a[0] = 0;
a[1] = 0;
a[2] = 0;

vector<double> b(3);
b[0] = 10;
b[1] = 0;
b[2] = 0;

vector<double> v = b - a; // 目的地へのベクトル
v = v / norm_2(v);        // 正規化

std::cout << v << std::endl; // [3](1,0,0)
```

Units

単位計算

```
using namespace boost::units;
using namespace boost::units::si;

quantity<length> x(1.5 * meter);          // 1.5m
quantity<length> y(120 * centi * meter); // 120cm
cout << x * y << endl; // 面積 : 1.8 m^2
```

Boost C++ Libraries

Unordered

ハッシュ表の連想コンテナ

```
boost::unordered_map<string, int> m;

// 辞書作成
m["Akira"]  = 3;
m["Millia"] = 1;
m["Johnny"] = 4;

// 要素の検索 : 見つからなかったらout_of_range例外が投げられる
int& id = m.at("Akira");
assert(id == 3);
```

Utility

その名の通りユーティリティ

```
class Window : private boost::noncopyable { // コピー禁止のクラスにする
  ...
};


struct functor {
  typedef int result_type;

  int operator()() const {}
};

// 関数(オブジェクト)の戻り値の型を取得
typedef boost::result_of<functor()>::type result_type;
BOOST_STATIC_ASSERT((boost::is_same<result_type, int>::value));
```

Boost C++ Libraries

Variant

指定した型を格納できるUnion

```
class my_visitor : public boost::static_visitor<int>
{
public:
  int operator()(int i) const
    { return i; }

  int operator()(const string& str) const
    { return str.length(); }
};

boost::variant<int, std::string> u("hello world");
cout << u; // "hello world"が出力される

int result = boost::apply_visitor(my_visitor(), u);
cout << result; // 11が出力される("hello world"の文字列長)
```

Boost C++ Libraries

Wave 1/2

C99, C++のプリプロセッサ

1. プリプロセッサを用意(wave.exeとする)

```
using namespace boost::wave;

std::ifstream file(argv[1]);
std::string source(std::istreambuf_iterator<char>(file.rdbuf()),
                   std::istreambuf_iterator<char>());

typedef context<std::string::iterator,
           cpplexer::lex_iterator<cpplexer::lex_token<> > > context_type;

context_type ctx(source.begin(), source.end(), argv[1]);

context_type::iterator_type first = ctx.begin();
context_type::iterator_type last  = ctx.end();

while (first != last) {
  std::cout << (*first).get_value();
  ++first;
}
```

Wave 2/2

2. C++のソースコードを用意する(a.cppとする)

```
#define HELLO "Hello World"

int main()
  { std::cout << HELLO << std::endl; }
```

3. a.cppをBoost.Waveでプリプロセス処理する

```
> wave.exe a.cpp

int main()
  { std::cout << “Hello World” << std::endl; }
```

Boost C++ Libraries

Xpressive

正規表現

```
// 文字列中の <ほげほげ> にマッチするものを検索
using namespace boost::xpressive;

std::string str = "The HTML tag <title> means that ...";
sregex rex = sregex::compile(“<[^>]+>”);

smatch result;
if (regex_search(str, result, rex)) {
  std::cout << "match : " << result.str() << std::endl;
}
```

```
match : <title>
```