

Boostライブラリー一周の旅

Ver 1.61.0 (Merge)

高橋 晶(Akira Takahashi)
faithandbrave@gmail.com

2016/07/23 Boost.勉強会 #20 東京

はじめに

この発表は、以下のような方を対象にして、

- Boostに興味はあるけど、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り
- Boostの全容を知りたい

Boost 1.61.0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とは

- 標準ライブラリに満足できなかった人たちが作っている、C++の準標準ライブラリ。
- Boostから標準ライブラリに、多くの機能が採用されている
- 普段のプログラミング全般で使える基本的なものから、専門的なものまで、いろいろなライブラリがある。
- ライセンスはBoost Software License 1.0
 - 無償で商用利用可能
 - 著作権表記の必要なし
 - ソースコードの改変自由

本日紹介するライブラリ

1. Accumulators	27.Filesystem	53.Metaparse	79.Scope Exit
2. Algorithm	28.Flyweight	54.Meta State Machine	80.Serialization
3. Align	29.Foreach	55.Move	81.Signals2
4. Any	30.Format	56.MPL	82.Smart Pointers
5. Array	31.Function	57.Multi Array	83.Sort
6. Asio	32.Function Types	58.Multi Index	84.Spirit
7. Assign	33.Fusion	59.Multiprecision	85.Static Assert
8. Atomic	34.Geometry	60.Numeric Conversion	86.String Algo
9. Bimap	35.GIL	61.Odeint	87.System
10.Bind	36.Graph	62.Operators	88.Test
11.Chrono	37.Hana	63.Optional	89.Thread
12.Circular Buffer	38.Heap	64.Overloaded Function	90.Timer
13.Compressed Pair	39.Identity Type	65.Parameter	91.Tokenizer
14.Compute	40.Interprocess	66.Phoenix	92.Tribool
15.Concept Check	41.Interval	67.Polygon	93.TTI
16.Container	42.Interval Container	68.Pool	94.Tuple
17.Conversion	43.Intrusive	69.Predef	95.Type Erasure
18.Convert	44.IO State Server	70.Preprocessor	96.Type Index
19 Coroutine	45.Iostreams	71.Property Map	97.Typeof
20.CRC	46.Iterators	72.Property Tree	98.uBLAS
21.Date Time	47.Lambda	73.Proto	99.Units
22.DLL	48.Local Function	74.Python	100.Unordered
23.Dynamic Bitset	49.Lockfree	75.Random	101.Uuid
24Endian	50.Log	76.Range	102.Variant
25.Enable If	51.Math	77.Ref	103.VMD
26.Exception	52.Member Function	78.ResultOf	104.Wave
			105.Xpressive

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
```

Algorithm 1/3

アルゴリズム集。文字列検索、C++11/14アルゴリズム、ユーティリティが含まれる。

```
std::string text =
    "the stick, and made believe to worry it: then Alice dodged behind a";
std::string pattern = "behind";

// BM法で文字列検索
decltype(text)::const_iterator it =
    boost::algorithm::boyer_moore_search(text.begin(), text.end(),
                                         pattern.begin(), pattern.end());

if (it != text.end()) std::cout << "found" << std::endl;
else                  std::cout << "not found" << std::endl;
```

found

Algorithm 2/3

アルゴリズム集。文字列検索、C++11/14アルゴリズム、ユーティリティが含まれる。

```
const std::vector<int> v = {2, 4, 6, 8, 10};

// 全ての値が偶数かを調べる
bool result = boost::algorithm::all_of(v, is_even);

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

true

Algorithm 3/3

アルゴリズム集。文字列検索、C++11/14アルゴリズム、ユーティリティが含まれる。

```
using boost::algorithm::clamp;  
  
// xを0~10の範囲に丸める : min(max(a, x), b)  
  
int x = 11;  
x = clamp(x, 0, 10); // x == 10  
  
int y = -1;  
y = clamp(y, 0, 10); // x == 0
```

Align

アライメント関係の情報を取得するメタ関数や
メモリアロケータなど。

```
// 16バイトアライメントでメモリ確保するアロケータを、  
// vectorで使用する  
std::vector<  
    char,  
    boost::alignment::aligned_allocator<char, 16>  
> v(100);
```

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。コンテナのインターフェースを使用できる。

```
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

ネットワークライブラリ。非同期処理全般を扱う。

```
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。

std::initializer_listを使用できない環境での代替。

```
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);
```

Atomic

C++11アトミックライブラリのC++03実装。

共有データ

```
int data;  
atomic<bool> ready(false); // アトミックなbool型変数
```

スレッド1

```
while (!ready.load(memory_order_acquire)) {} // 書き込まれるまで待機  
std::cout << data << std::endl; // 3が出力されることが保証される
```

スレッド2

```
data = 3;  
ready.store(true, memory_order_release); // 書き込み
```

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
```

Chrono

時間処理のためのライブラリ。

C++11標準ライブラリに導入されたものと、その拡張。

```
// 500ナノ秒遅延する
namespace chrono = boost::chrono;

auto go = chrono::steady_clock::now() + chrono::nanoseconds(500);
while (chrono::steady_clock::now() < go)
    ;
```

様々な時間の単位と、いくつかの特性をもった時計クラスが提供される。
CPU時間を使う拡張もある。

Circular Buffer

循環バッファ。

バッファがいっぱいになつ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
```

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();
```

Compute

OpenCLをベースとした、マルチコアCPUやGPGPUを扱うライブラリ。
boost::compute名前空間にあるデータ構造とアルゴリズムが、デバイス用のもの。ホストのデータ構造とやりとりできる。設計はThrust風。

```
// デバイスのセットアップ…

std::vector<int> host_data = { 1, 3, 5, 7, 9 };
compute::vector<int> device_vector(5, context);

// ホスト環境からデバイス環境にデータをコピー
compute::copy(
    host_data.begin(),
    host_data.end(),
    device_vector.begin(), queue
);
```

Concept Check 1/3

テンプレートパラメータの制約

(C++0xで導入されなかったConcept言語機能のライブラリ版)

```
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());
```

Concept Check 1/3

Concept Checkを使わなかった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 1/3

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
```

...

かなりよくなつた

Container

標準コンテナのBoost実装。

placement insertやmoveなどの最新仕様が提供される。

```
struct Person {  
    int id;  
    std::string name;  
    Person() {}  
    Person(int id, const std::string& name) : id(id), name(name) {}  
};  
  
boost::container::vector<Person> v;  
  
// これまで通りのpush_backだが、一時オブジェクトならmoveされる  
v.push_back({1, "Alice"});  
  
// 関数内部でコンストラクタを呼び出すplacement insert  
v.emplace_back(2, "Bob");
```

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);
```

Convert 1/3

boost::lexical_cast()を置き換えて使用できる、数値と文字列の型変換ライブラリ。

変換失敗時の挙動、基数や精度、フォーマットなどを設定できる。

戻り値の型はboost::optional<T>。

```
namespace cnv = boost::cnv;

// 文字列"123"をint型に変換する
// コンバータにはcstreamを使用する
cnv::cstream converter;
boost::optional<int> result =
    boost::convert<int>("123", converter);
```

コンバータクラスはいくつか用意されており、現在正式にサポートされているのは、lexical_castと(c|w)streamの2つ。
コンバータを別定義できるので、文字コードの変換にも利用できる。

Convert 2/3

変換失敗時のエラーハンドリングは基本的に、戻り値であるboost::optional<T>オブジェクトに対して行う。

```
boost::cnv::cstream converter;

// 変換後にoptionalの中身を取り出す。
// 変換失敗時はboost::bad_optional_access例外が送出される
int result2 = boost::convert<int>("123", converter).value();

// 変換失敗時に-1が返される
int result3 = boost::convert<int>("xxx", converter).value_or(-1);
```

Convert 3/3

基数や精度、フォーマットの設定は、標準ライブラリのマニピュレータと、
Boost.Convertが定義しているパラメータの、2種類が使用できる。
マニピュレータに指定は、コンバータの関数呼び出し演算子を使用する。

```
// 標準のマニピュレータ
cnv::cstream converter;
int result1 = boost::convert<int>(
    "ff",
    converter(std::hex)(std::skipws) // 16進数、スペースを無視
).value(); // result == 255

// Boost.Convertのマニピュレータ
int result2 = boost::convert<int>(
    "ff",
    converter(arg::base = cnv::base::hex) // 16進数
).value(); // result == 255
```

Coroutine

処理の中斷と再開を制御する、コルーチンのライブラリ。

```
using namespace boost::coroutines;

void f(asymmetric_coroutine<void>::push_type& yield)
{
    for (int i = 0; i < 10; ++i) {
        std::cout << "a ";
        yield(); // 処理を中斷
    }
}

asymmetric_coroutine<void>::pull_type c(f);
for (int i = 0; i < 10; ++i) {
    std::cout << "b ";
    c(); // 中断したところから処理を再開
}
```

a b a b a 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);
```

DLL

DLLやSOといった形式になっているC++ライブラリを扱うライブラリ。
インポート、エクスポート、関数呼び出しなどの機能をサポートする

```
// インポート側
auto cpp11_func = dll::import<int(std::string&&)>(
    path_to_shared_library, "i_am_a_cpp11_function");
// 関数呼び出し
cpp11_func("hello");
```

```
// エクスポート側
#define API extern "C" BOOST_SYMBOL_EXPORT
namespace some_namespace {
    API int i_am_a_cpp11_function(std::string&& param) noexcept;
}
```

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
```

Endian

エンディアン操作のライブラリ。

エンディアン指定の算術型や、エンディアン変換の関数など。

```
using namespace boost::endian;
```

```
little_uint8_t a; // リトルエンディアンの8ビット符号なし整数型  
big_uint8_t b; // ビッグエンディアンの(以下略)
```

```
std::uint8_t x = 0;  
std::ifstream file(...);  
file.read(&x, sizeof(uint8_t));
```

```
x = big_to_native(x); // ビッグエンディアンのデータを、  
// その環境のエンディアンに変換。
```

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());
```

Foreach

foreach文のマクロ。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;  
}
```

```
abs  
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
```

Function

汎用関数オブジェクト。

テンプレート引数は、関数のシグニチャ。

```
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);
```

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。

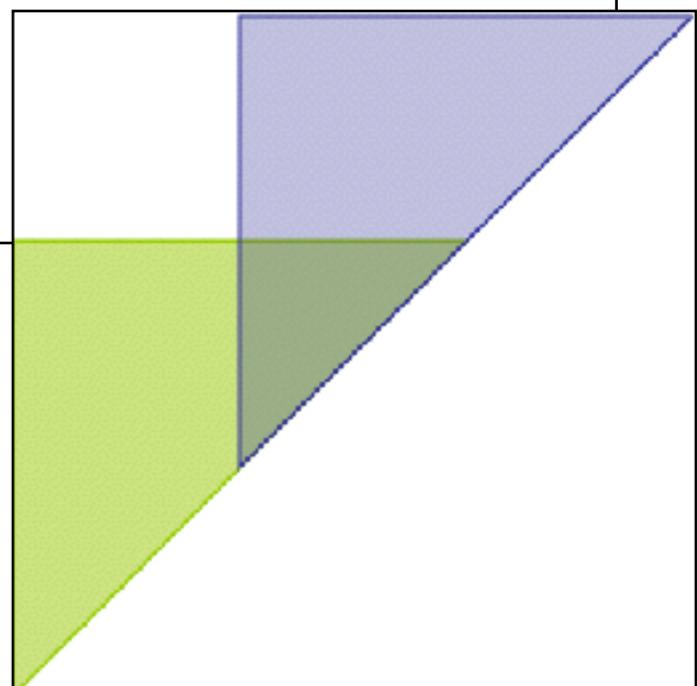
```
struct disp {  
    template <class T>  
    void operator()(T x) const { cout << x << endl; }  
};  
  
using namespace boost::fusion;  
  
// コンパイル時のタプル(型リスト)操作：一番後ろの型を取り除く  
typedef vector<int, double, string, string>           typelist;  
typedef result_of::pop_back<typelist>::type          unique_typelist;  
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());
```

Geometry

計算幾何のライブラリ。

N次元の点、線分、ポリゴンなどのモデルと、
それらに対するアルゴリズムが提供される。

```
polygon a, b;  
geometry::exterior_ring(a) =  
    assign::list_of<point>(0, 0)(3, 3)(0, 3)(0, 0);  
  
geometry::exterior_ring(b) =  
    assign::list_of<point>(1.5, 1.5)(4.5, 4.5)(1.5, 4.5)(1.5, 1.5);  
  
// 2つのポリゴンが交わっているか  
const bool result = geometry::intersects(a, b);  
BOOST_ASSERT(result);
```



GIL

画像処理ライブラリ。

```
using namespace boost::gil;

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

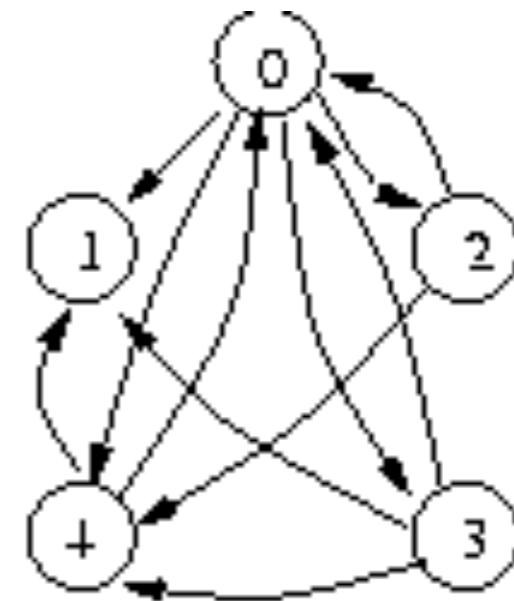
// 100x100にリサイズ
rgb8_image_t square (100, 100);
resize_view(const_view(img),
            view(square),
            bilinear_sampler());
jpeg_write_view("out-resize.jpg", const_view(square));
```

Graph

グラフ構造とアルゴリズムのライブラリ。

```
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);
```



Hana 1/2

C++14時代のメタプログラミングライブラリ。
FusionとMPLを合わせたもの。
値のシーケンスと型のシーケンス両方への操作を一様にできる。

```
// 値のシーケンスを扱う例
auto xs = hana::make<hana::tuple_tag>(1, 3.14, "hello");

// タプルの全ての要素を文字列に変換
auto ys = hana::transform(xs, [](auto x) {
    return boost::lexical_cast<std::string>(x);
});

hana::for_each(ys, [](const std::string& x) {
    std::cout << x << std::endl;
});
```

Hana 2/2

C++14時代のメタプログラミングライブラリ。
FusionとMPLを合わせたもの。
値のシーケンスと型のシーケンス両方への操作を一様にできる。

```
// 型のシーケンスを扱う例
auto xs = hana::tuple_t<int, double, std::string>;

// 算術型のみ抽出する
auto ys = hana::filter(xs, [](auto x) {
    return hana::traits::is_arithmetic(x);
});

static_assert(hana::to_tuple(ys) == hana::tuple_t<int, double>);
```

Heap 1/2

優先順位付きキューのデータ構造。

```
boost::heap::fibonacci_heap<int> que; // フィボナッチヒープ  
  
que.push(3);  
que.push(1);  
que.push(4);  
  
while (!que.empty()) {  
    std::cout << que.top() << std::endl;  
    que.pop();  
}
```

```
4  
3  
1
```

Heap 2/2

Boost.Heapの特徴：

- 要素の修正ができる (Mutability)
- イテレータを持っている (Iterators)
- マージできる (Mergable)
- 安定 (Stability)

Identity Type

関数マクロに渡す引数で、カンマを付けられるようにする。

```
std::map<int, int> m = {{1, 3}, {2, 4}};  
  
BOOST_FOREACH(  
    BOOST_IDENTITY_TYPE((std::map<int, int>))::const_reference x, m) {  
    std::cout << x.first << "," << x.second << std::endl;  
}
```

```
1,3  
2,4
```

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

区間演算

```
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]
```

Interval Container (ICL)

区間演算のコンテナを提供するライブラリ。

```
typedef std::set<string> guests;
interval_map<ptime, guests> party;
party += make_pair(interval<ptime>::right_open(
    time_from_string("20:00"),
    time_from_string("22:00")),
    make_guests("Mary"));

party += make_pair(interval<ptime>::right_open(
    time_from_string("21:00"),
    time_from_string("23:00")),
    make_guests("Harry"));
```

```
[20:00, 21:00)->{"Mary"}
[21:00, 22:00)->{"Harry", "Mary"} // 時間帯が重なっていたら集約される
[22:00, 23:00)->{"Harry"}
```

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));
}
```

IO State Saver

ストリームの状態管理。

```
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)
```

lostreams

拡張IO Streamライブラリ。
パイプ演算によるフィルタや圧縮など。

```
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;
```

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
```

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
```

以下の式では、

```
cout << _1 << ' ';
```

以下のような関数オブジェクトが作られる：

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

Local Function

ローカル関数を定義する。

```
int main()
{
    int sum = 0;

    void BOOST_LOCAL_FUNCTION(bind& sum, int x) {
        sum += x;
    } BOOST_LOCAL_FUNCTION_NAME(add);

    const std::vector<int> v = {1, 2, 3, 4, 5};
    boost::for_each(v, add);

    std::cout << sum << std::endl;
}
```

Lockfree

Boost.Atomicベースの、ロックフリーデータ構造ライブラリ。
キュー、スタック、優先順位付きキューの実装がある。

```
lockfree::queue<int> que(128);

void producer() {
    for (int i = 0;; ++i) {
        while (!que.push(i)) {}
    }
}

void consumer() {
    for (;;) {
        int x = 0;
        if (que.pop(x))
            std::cout << x << std::endl;
    }
}
```

Log

ロギングライブラリ。ログレベルの設定、フォーマット、ファイルサイズや日付によるローテーションなど。

```
using namespace boost::log;
add_file_log("log.txt");

// infoレベル以上を出力し、それ以外は捨てる
core::get()->set_filter(
    trivial::severity >= trivial::info
);

BOOST_LOG_TRIVIAL(debug) << "デバッグメッセージ";
BOOST_LOG_TRIVIAL(info)  << "情報メッセージ";
BOOST_LOG_TRIVIAL(error) << "エラーメッセージ";
BOOST_LOG_TRIVIAL(fatal) << "致命的なエラーメッセージ";
```

Math

数学関数。

```
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
```

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));
```

Metaparse 1/3

コンパイル時の構文解析ライブラリ。

コンパイル時文字列クラスも付いている。

printfのフォーマットや正規表現などをコンパイル時に検証するために
使用できる。

```
// コンパイル時文字列
using hello1 = string<'H', 'e', 'l', 'l', 'o'>;
using hello2 = BOOST_METAPARSE_STRING("Hello");

static_assert(
    std::is_same_v<hello1, hello2>::type::value
);
```

Metaparse 2/3

コンパイル時の構文解析ライブラリ。

コンパイル時文字列クラスも付いている。

printfのフォーマットや正規表現などをコンパイル時に検証するために
使用できる。

```
// コンパイル時に、文字列を整数に変換
using digit_value = transform<digit, util::digit_to_int>;
static_assert(
    get_result<
        digit_value::apply<BOOST_METAPARSE_STRING("0"), start>
    >::type::value == 0
);
```

Metaparse 3/3

コンパイル時の構文解析ライブラリ。

コンパイル時文字列クラスも付いている。

printfのフォーマットや正規表現などをコンパイル時に検証するために
使用できる。

```
// 演算式を解析
using expr = sequence<token<int_>, token<lit_c<'+ '>>, token<int_>>;
using parser = build_parser<expr>;

static_assert(
    !is_error<
        parser::apply<BOOST_METAPARSE_STRING("11 + 2")>
    >::type::value
);
```

Meta State Machine (MSM) 1/2

状態マシンライブラリ。状態遷移表を直接記述する。

```
namespace msm = boost::msm;
struct Active : msm::front::state<> {};
struct Stopped : msm::front::state<> {};
struct StartStopEvent {};
struct ResetEvent {};

struct StopWatch_ : msm::front::state_machine_def<StopWatch_> {
    typedef Stopped initial_state;
    struct transition_table : boost::mpl::vector<
        //          Start   Event           Next
        _row<Active, StartStopEvent, Stopped>,
        _row<Active, ResetEvent,      Stopped>,
        _row<Stopped, StartStopEvent, Active>
    > {};
};

typedef msm::back::state_machine<StopWatch_> Stopwatch;
```

Meta State Machine (MSM) 2/2

状態マシンライブラリ。状態遷移表を直接記述する。

```
int main()
{
    StopWatch watch;

    watch.start();
    watch.process_event(StartStopEvent()); // stop -> run
    watch.process_event(StartStopEvent()); // run  -> stop
    watch.process_event(StartStopEvent()); // stop -> run
    watch.process_event(ResetEvent());    // run  -> stop
}
```

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
```

Move

ムーブセマンティクスのC++03実装。
一時オブジェクトのコストを軽減する。

```
template <class T>
void swap(T& a, T& b)
{
    T tmp(boost::move(a));
    a = boost::move(b);
    b = boost::move(tmp);
}
```

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++

Multiprecision

多倍長演算ライブラリ。無限長の整数などを扱える。

```
// 100の階乗を計算する
cpp_int x = 1;
for(std::size_t i = 1; i <= 100; ++i)
    x *= i;

std::cout << x << std::endl;
```

```
9332621544394415268169923885626670049071596826438162146859296389521759
9993229915608941463976156518286253697920827223758251185210916864000000
000000000000000000
```

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;
}
```

Odeint

常微分方程式を解くためのライブラリ。カオス理論、振り子の演算など。
以下は、ローレンツ方程式の例。

```
const double sigma = 10.0;
const double R = 28.0;
const double b = 8.0 / 3.0;
typedef boost::array< double , 3 > state_type;

void lorenz( const state_type &x , state_type &dxdt , double t ) {
    dxdt[0] = sigma * ( x[1] - x[0] );
    dxdt[1] = R * x[0] - x[1] - x[0] * x[2];
    dxdt[2] = -b * x[2] + x[0] * x[1];
}

void write_lorenz( const state_type &x , const double t )
{ cout << t << '\t' << x[0] << '\t' << x[1] << '\t' << x[2] << endl; }

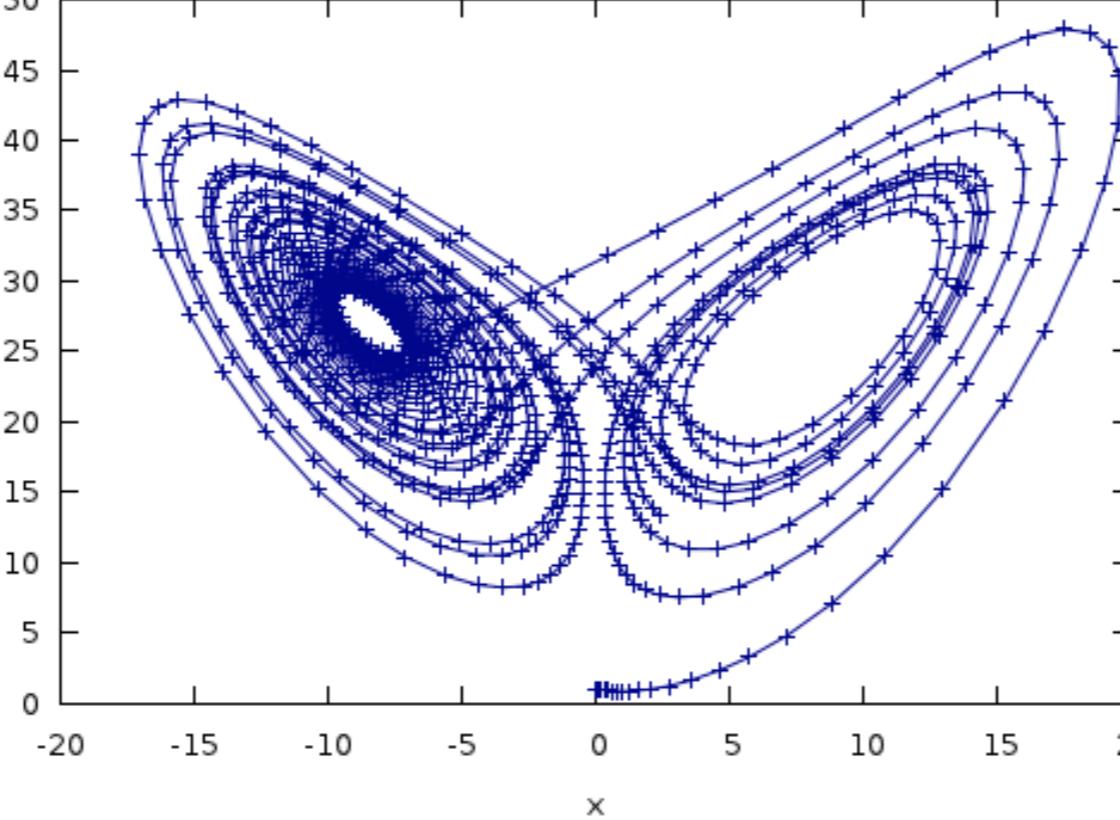
state_type x = { 10.0 , 1.0 , 1.0 }; // initial conditions
integrate( lorenz , x , 0.0 , 25.0 , 0.1 , write_lorenz );
```

Odeint

常微分方程式を解くためのライブラリ。カオス理論、振り子の演算など。
以下は、ローレンツ方程式の例。

```
const double sigma = 10.0;
const double P - 20.0.
const double I
typedef boost

void lorenz( , double t ) {
    dxdt[0] =
    dxdt[1] =
    dxdt[2] =
}
void write_lo
{ cout << t << endl;
state_type x = { 10.0 , 1.0 , 1.0 }; // initial conditions
integrate( lorenz , x , 0.0 , 25.0 , 0.1 , write_lorenz );
```



Operators

関連する演算子の自動生成。

```
class person : boost::less_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の、統一的な表現を提供するライブラリ。
エラー値が-1だったりヌルポインタだったりするのを統一する。

```
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 *it;           // 見つかったら有効な値を返す
    return boost::none; // 見つからなかつ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;
```

Overloaded Function

複数の関数から、オーバーロードする関数オブジェクトを作る。

```
const std::string& identity_s(const std::string& s) { return s; }
int identity_n(int x) { return x; }
double identity_d(double x) { return x; }

boost::overloaded_function<
    const std::string& (const std::string&),
    int (int),
    double (double)
> identity(identity_s, identity_n, identity_d);

std::string s = "hello"; s = identity(s);
int n = 2; n = identity(n);
double d = 3.14; d = identity(d);
```

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
```

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
```

Predef

コンパイラ、アーキテクチャ、OS、標準ライブラリなどの情報を取得するライブラリ。

これらのマクロは必ず定義されるので、if文で使ってもいい。

```
// GCCかどうか
#if BOOST_COMP_GNUC
    #if BOOST_COMP_GNUC >= BOOST_VERSION_NUMBER(4,0,0)
        const char* the_compiler = "バージョン4以上のGNU GCC,"
    #else
        const char* the_compiler = "バージョン4未満のGNU GCC"
    #endif
#else
    const char* the_compiler = "GNU GCCではない"
#endif
```

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(たとえば可変引数)

```
#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;
}

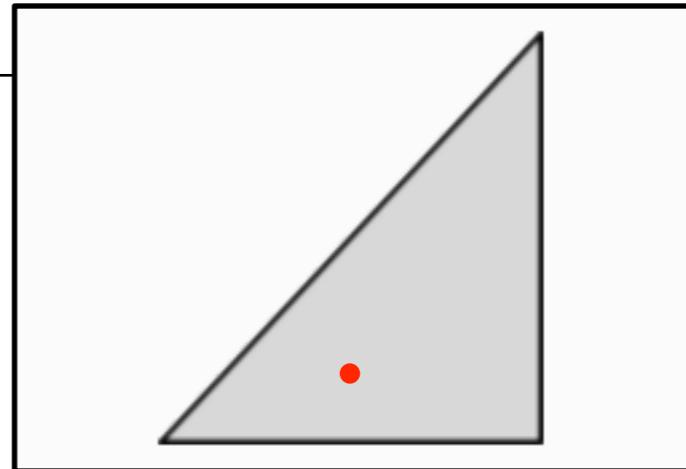
assert(add(1, 2, 3) == 6)
```

Polygon

平面多角形(2D)のアルゴリズムを提供するライブラリ。
以下は、三角形の内外判定。

```
#include <boost/polygon/polygon.hpp>
namespace polygon = boost::polygon;

int main()
{
    const std::vector
```



Property Map

インターフェースのマッピング。Boost.Graphで使われている。
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);
```

Property Tree 1/4

汎用的な、木構造をもつデータのプロパティ管理。
XML、JSON、INIファイルのパーサーを提供している。

全てのデータは、
`boost::property_tree::ptree`型
に対して操作を行う。

値の取得には、
失敗時に例外を投げる`ptree::get<T>()`と
`boost::optional`を返す`ptree::get_optional<T>()`
が用意されている。

Property Tree 2/4

XMLの読み込み、要素と属性の取得。

XMLパーサーには、RapidXmlを採用している。

```
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");
```

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

Property Tree 3/4

JSONの読み込み、データの取得。

```
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");
```

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

Property Tree 4/4

INIの読み込み、データの取得。

```
using namespace boost::property_tree;  
  
ptree pt;  
read_ini("test.ini", pt);  
  
const int    value = pt.get<int>("Data.Value");  
const string& str  = pt.get<string>("Data.Str");
```

[Data]
Value = 314
Str = Hello

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");
```

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を作成される)

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

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

3. Python側で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
```

Random

乱数ライブラリ。

乱数生成器と分布を分けて指定できるのが特徴。

```
// シード生成：非決定的な乱数生成器 (random_device)
// 疑似乱数生成器：メルセンヌツイスター法 (mt19937)
// 分布方法：一様整数分布[1, 10] (uniform_int_distribution)
using namespace boost::random;

random_device seed_gen;
mt19937 engine(seed_gen());
uniform_int_distribution<int> dist(1, 10);

for (int i = 0; i < 10; ++i) {
    int result = dist(engine);
    cout << result << endl;
}
```

Range

範囲に対するアルゴリズムと、Rangeアダプタ。

```
std::vector<int> v;  
  
// Rangeアルゴリズム : イテレータの組ではなく範囲を渡す  
boost::sort(v);  
boost::for_each(v, f);  
  
// Rangeアダプタ  
using namespace boost::adaptors;  
boost::for_each(v | filtered(p) | transformed(conv), f);
```

- Rangeアルゴリズム : 標準アルゴリズムのRange版
- Rangeアダプタ : 遅延評価され、合成可能な範囲操作

Ref

参照ラッパ

関数テンプレートのパラメータが「`T x`」になっているときに、引数として変数を渡すと、参照ではなくコピーで渡されてしまう。`boost::ref()`関数でラップすると、明示的に参照で渡せる。

```
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
```

Result Of

関数(オブジェクト)の戻り値の型を取得するメタ関数。

```
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));
```

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、変数を参照キャプチャ。

(variable)で、変数をコピーキャプチャ。

```
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
};
```

Serialization

クラス情報をシリアル化／デシリアル化する。

```
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("./person.xml");  
boost::archive::xml_oarchive oarchive(ofs); // XMLシリアル化  
  
Person person;  
oarchive << boost::serialization::make_nvp("Root", person);
```

Signals 2

シグナル・スロット。オブジェクト間のイベントを管理する。

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

Smart Pointers

スマートポインタ。

リソースを使い終わったら、自動的に一度だけ破棄してくれる。

```
void foo()
{
    // スコープを抜けたら解放されるスマートポインタ
    boost::shared_ptr<hoge> h(new hoge());

    h->bar();
} // ここでdeleteされる
```

Sort

範囲を並び替えるアルゴリズムのライブラリ。

巨大な配列をソートする際にstd::sort()よりも高速になる、
spreadsort()が含まれている。

```
using namespace boost::sort::spreadsort;

std::vector<int> v; // N >= 10,000
spreadsort(v.begin(), v.end());
```

この関数に指定できるのは、以下の型の範囲のみ：

- 整数型
- 浮動小数点数型
- 文字列型

Spirit

構文解析。

```
// "(1.0, 2.0)"という文字列をパースして、結果をdoubleの組に格納する
string input("(1.0, 2.0)");
string::iterator it = input.begin();

pair<double, double> result;
if (qi::parse(it, input.end(),
    '(' >> qi::double_ >> ", " >> qi::double_ >> ')',
    result)) {
    cout << result.first << ", " << result.second << endl;
}
```

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ために使用する。

```
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！
```

String Algo

文字列のアルゴリズム。

```
string str = "hello world ";

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

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

System

各OSのエラーコードをラップして汎用化したもの。

```
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

自動テストのライブラリ

```
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);
}

using namespace boost::unit_test_framework;
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

Test v3 (1.59.0)

Boost.Testがバージョン3にメジャーアップデートした。

汎用的なテストマクロBOOST_TESTが追加された(Power Assert)。

パラメタライズドテストに対応した。ドキュメントが読みやすくなった。

```
BOOST_AUTO_TEST_CASE(equal_test)
{
    int a = 1;
    int b = 2;
    BOOST_TEST(a == b);
}
```

```
Running 1 test case...
```

```
main.cpp:8: error: in "equal_test": check a == b has failed [1 != 2]
```

```
*** 1 failure is detected in the test module "example"
```

Thread

スレッドを使用した並行プログラミングのライブラリ。

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

int main()
{
    boost::thread t(hello); // 関数hello()をバックグラウンドスレッドで実行
    t.join(); // スレッドの終了を待機する
}
```

Timer

時間計測。

```
boost::timer::cpu_timer timer; // 時間計測を開始

for (long i = 0; i < 100000000; ++i) {
    std::sqrt(123.456L); // 時間のかかる処理
}

std::string result = timer.format(); // 結果文字列を取得する
std::cout << result << std::endl;
```

```
5.636670s wall, 5.600436s user + 0.000000s system = 5.600436s CPU (99.4%)
```

Tokenizer

トークン分割。

ポリシーによって、さまざまな分割方法を設定できる(たとえばCSV)

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

for_each(tok.begin(), tok.end(), [](const string& t) {
    cout << t << endl;
});
```

```
This
is
a
pen
```

Tribool

3値bool。真・偽・不定(NULL相当)の3つの値を持つ。
データベースとのマッピングに使用できる。

```
using namespace boost::logic;

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

TTI (Type Traits Introspection)

型がどんな情報を持っているかコンパイル時に調べるライブラリ。

```
// メンバ関数mf()を持っているか判定するメタ関数を生成する
BOOST_TTI_HAS_MEMBER_FUNCTION(mf)

struct X {
    int mf(int, int);
};
```

型Xが、intを2つ受け取り、intを返すメンバ関数mf()を持っているか

```
constexpr bool b = has_member_function_mf<
    X,
    int,
    boost::mpl::vector<int, int>
>::value;
```

Tuple

タプル。

`std::pair` は 2 つの値の組だが、タプルは 3 つ以上の値も設定できる。

```
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)
```

Type Erasure 1/2

コンセプトで実行時多相性を表現するライブラリ。

「できること」を列挙して設定したany型に、それが可能なあらゆる型のオブジェクトを実行時に代入して使用できる。

```
using namespace boost::type_erasure;
any<
    boost::mpl::vector<
        copy_constructible<>, // コピー構築できること
        incrementable<>,    // インクリメントできること
        ostreamable<>       // ストリーム出力できること
>
> x(10); // int値をコピー(要件を満たしている型ならなんでもOK)
++x;
std::cout << x << std::endl; // 「11」が出力される
```

Type Erasure 2/2

コンセプトは、自分で定義できる。

```
// 1引数をとるpush_back()メンバ関数を持っていること、  
// というコンセプトを定義する。  
BOOST_TYPE_ERASURE_MEMBER((has_push_back), push_back, 1)  
  
// intを引数にとり、voidを返すpush_backを持っている、  
// コンテナへの参照を受け取って操作する  
void append_many(any<has_push_back<void(int)>, _self&> container)  
{  
    for(int i = 0; i < 10; ++i)  
        container.push_back(i);  
}
```

Type Index

std::type_info／std::type_indexのBoost版。
RTTIの無効化や、型のデマングル名などに対応している。

```
using boost::typeindex::type_id;
std::cout << type_id<int>().raw_name() << std::endl;
std::cout << type_id<int>().pretty_name() << std::endl;
```

i
int

(GCCの場合)

Typeof

型推論。

C++11のautoとdecltypeをエミュレーションする。

```
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++11では以下のようになる：

```
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;
```

uBLAS

ベクトルや行列といった、線形代数のライブラリ。

```
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

单位演算

```
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 m2
```

Unordered

ハッシュ表の連想コンテナ。

C++11で標準ライブラリに導入された。

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

// 辞書作成
m["Alice"] = 3;
m["Bob"] = 1;
m["Carol"] = 4;

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

Uuid

ユニークIDの作成。

COMとか、分散環境での情報の識別とかで使われることが多い。

```
using namespace boost::uuids;

// 擬似乱数生成器でのUUID生成。デフォルトはmt19937
uuid u1 = random_generator()();

// 文字列からUUID生成
uuid u2 = string_generator()("0123456789abcdef0123456789abcdef");

cout << u1 << endl;
cout << u2 << endl;
```

```
31951f08-5512-4942-99ce-ae2f19351b82
01234567-89ab-cdef-0123-456789abcdef
```

Variant

指定した型を格納できる共用体。

```
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 << endl; // "hello world"が出力される

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

VMD (Variadic Macro Data Library)

可変引数マクロを使用した、プリプロセッサメタプログラミングのライブラリ。Boost.Preprocessorを強化するためのもの。
各種データ型と、それをテスト・解析する機能が提供される。

```
#define SEQ 1 2 3
#define SEQ_SIZE BOOST_VMD_SIZE(SEQ)

void f(int a, int b, int c)
{ std::cout << a << " " << b << " " << c << std::endl; }

// 要素数を取得
std::cout << SEQ_SIZE << std::endl; // 3

// シーケンスをカンマ区切りパラメータに変換
f(BOOST_VMD_ENUM(SEQ)); // 「1, 2, 3」に変換される
```

Wave 1/2

CとC++のプリプロセッサ。

1. プリプロセッサを用意する(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; }
```

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>