Boost.Python

Dave Abrahams

翻訳にあたって

- 本書は Boost.Python ドキュメントの日本語訳です。原文書のバージョンは翻訳時の最新である 1.55.0 です。
- 原文の誤りは修正したうえで翻訳しました。
- 外部文書の表題等は英語のままにしてあります。
- Python のサイトへのリンクは可能な限り日本語サイトへ変更しました。
- 原文に含まれているローカルファイルへのハイパーリンクは削除したか、Boost のサイトへのリンクに変更しました。
- ファイル名、ディレクトリ名は `pathname` のように記します。
- その他、読みやすくするためにいくつか書式の変更があります。
- 翻訳の誤り等は exeal に連絡ください。

概要

Boost.Python のバージョン 2 へようこそ。Boost.Python は、C++と Python プログラミング言語間のシームレスな相互運用を可能にする C++ライブラリである。新バージョンは、より便利で柔軟なインターフェイスを備えるために完全に一から書き直した。加えて以下の項目をサポートする機能を追加した：

- 参照とポインタ
- 大域に登録した型変換
- モジュール間の自動的な型変換
- 効率的な関数多重定義
- C++例外の Python への変換
- 既定の引数
- キーワード引数
- C++における Python オブジェクトの操作
- C++のイテレータを Python のイテレータとしてエクスポート
- ドキュメンテーション文字列

これらの機能の開発は、Boost Consulting に対する Lawrence Livemore National Laboratories の一部助成と、Lawrence Berkeley

1 訳注　原著のライセンス：*"Distributed under the Boost Software License, Version 1.0. (See accompanying file LICENSE_1_0.txt or copy at http://www.boost.org/LICENSE_1_0.txt)"*。本稿のライセンスも同様とします。

National Laboratories の Computational Crystallography Initiative に提供を受けた。

チュートリアル

クイックスタート

Boost.Python ライブラリは Python と C++間のインターフェイスのためのフレームワークである。C++のクラス、関数、オブジェクトをすばやくシームレスに Python へエクスポートでき、また逆に Python から C++へエクスポートできる。特別なツールは必要なく、あなたのコンパイラだけで可能だ。このライブラリは C++インターフェイスを非侵入的にラップするよう設計されており、C++コードを変更する必要は一切ない。このため、サードパーティ製ライブラリを Python へエクスポートするには Boost.Python が最適である。ライブラリは高度なメタプログラミング技術を使ってその構文を単純化しており、コードのラッピングは宣言的なインターフェイス定義言語（IDL）のような見た目になっている。

Hello World

C/C++の伝統に従い「hello, world」から始めるとしよう。

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

この C++関数は、次の Boost.Python ラッパを書くことで Python へエクスポートできる。

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

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

これですべてである。あとはこれを共有ライブラリとしてビルドすると、生成した DLL が Python から可視となる。以下は Python のセッション例である。

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

次回、「Hello World モジュールのビルド」。

Hello World のビルド

始めから終わりまで

まず最初は Hello World モジュールをビルドして Python で使ってみることだろう。本節でそのためのステップを明らかにする。あらゆる Boost ディストリビューションに付属するビルドツールである **bjam** を使用する。

> **注意**
>
> **bjam を使用せずにビルドする**
>
> 当然 bjam 以外にモジュールをビルドする方法はある。ここに書いていることが「唯一の方法」というわけではない。bjam の他にビルドツールは存在する。
>
> しかしながら Boost.Python のビルドには bjam が適していると記しておく。セットアップを失敗させる方法はたくさんある。経験から言えば「Boost.Python がビルドできない」という問題の 9 割は、他のツールを使用することを余儀なくされた人から寄せられた。

細かいことは省略する。ここでの目的は Hello World モジュールを簡単に作成して Python で走らせることである。Boost.Python のビルドについて完全なリファレンスは「ビルドとテスト」を見るとよい。この短いチュートリアルが終わったら DLL のビルドが完了して Python のプログラムで拡張が走っているはずである。

チュートリアルの例はディレクトリ `/libs/python/example/tutorial` にある。以下のファイルがある。

- `hello.cpp`
- `hello.py`
- `Jamroot`

`hello.cpp` ファイルは C++の Hello World 例、`Jamroot` は DLL をビルドする最小限の bjam スクリプトである。そして `hello.py` は `hello.cpp` の拡張を使用する Python プログラムである。

何よりもまず bjam の実行可能ファイルを boost ディレクトリか、`bjam` をコマンドラインから実行できるパスに置いておく。ほとんどのプラットフォームでビルド済み Boost.Jam 実行可能ファイルが利用できる。bjam 実行可能ファイルの完全なリストがここにある。

Jam ろう！

最小限の Jamroot ファイルを `/libs/python/example/tutorial/Jamroot` に置いておく。そのままファイルをコピーして `use-project boost` の部分を Boost のルートディレクトリに設定すればよい。

必要なことはこの Jamrules ファイルのコメントに書いてある。

bjam を起動する

オペレーティングシステムのコマンドラインインタープリタから bjam を起動する。

では、始めるとしよう。

`user-config.jam` という名前のファイルをホームディレクトリに置いてツールを調整する。Windows の場合、コマンドプロンプト

ウィンドウで次のようにタイプするとホームディレクトリがわかる。

```
ECHO %HOMEDRIVE%%HOMEPATH%
```

ファイルには少なくともコンパイラと Python のインストールについてのルールを書いておく必要がある。Windows の場合は例えば以下のとおり:

```
#  Microsoft Visual C++の設定
using msvc : 8.0 ;

#  Python の設定
using python : 2.4 : C:/dev/tools/Python ;
```

1 番目のルールで MSVC 8.0 とその関連ツールを使用することを bjam に設定している。2 番目のルールは Python についての設定であり、Python のバージョンと場所を指定している。上の例では `C:/dev/tools/Python` に Python をインストールした想定である。Python を正しく「標準的に」インストールした場合はこの設定は不要である。

ここまで来れば準備は整った。チュートリアルの `hello.cpp` と `Jamroot` が置いてある `libs/python/example/tutorial` に `cd` で移動するのを忘れないように。

```
bjam
```

これでビルドが始まり、

```
cd C:\dev\boost\libs\python\example\tutorial
bjam
...patience...
...found 1101 targets...
...updating 35 targets...
```

最終的に例えば以下のように表示される。

```
Creating library path-to-boost_python.dll
   Creating library /path-to-hello_ext.exp/
**passed** ... hello.test
...updated 35 targets...
```

すべて問題なければ、DLL がビルドされ Python のプログラムが走るはずである。

さあ、楽しんでいただきたい！

クラスのエクスポート

では C++クラスを Python へエクスポートしよう。

エクスポートすべき C++クラス・構造体を考えよう。

```
struct World
{
    void set(std::string msg) { this->msg = msg; }
    std::string greet() { return msg; }
    std::string msg;
};
```

相当する Boost.Python ラッパを書いて Python へエクスポートできる。

```
#include <boost/python.hpp>
using namespace boost::python;

BOOST_PYTHON_MODULE(hello)
{
    class_<World>("World")
        .def("greet", &World::greet)
        .def("set", &World::set)
    ;
}
```

上記のようにメンバ関数 `greet` および `set` をエクスポートする C++クラスラッパを書いた。このモジュールを共有ライブラリとしてビルドすると、Python 側から `World` クラスが使用できるようになる。次に示すのは Python のセッション例である。

```
>>> import hello
>>> planet = hello.World()
>>> planet.set('howdy')
>>> planet.greet()
'howdy'
```

コンストラクタ

前回の例では明示的なコンストラクタが登場しなかった。`World` はプレーンな構造体として宣言したので、暗黙のデフォルトコンストラクタとなっていた。Boost.Python は既定ではデフォルトコンストラクタをエクスポートするので、以下のように書けた。

```
>>> planet = hello.World()
```

デフォルトでないコンストラクタを使ってクラスをラップしたい場合もあるだろう。前回の例をビルドする。

```
struct World
{
    World(std::string msg): msg(msg) {} // コンストラクタを追加した
    void set(std::string msg) { this->msg = msg; }
    std::string greet() { return msg; }
    std::string msg;
};
```

これで `World` にデフォルトコンストラクタはなくなった。前回のラップコードは、ライブラリをエクスポートするところでコンパイルに失敗するだろう。代わりにエクスポートしたいコンストラクタについて `class_<World>`に通知しなければならない。

```
#include <boost/python.hpp>
using namespace boost::python;

BOOST_PYTHON_MODULE(hello)
{
    class_<World>("World", init<std::string>())
        .def("greet", &World::greet)
        .def("set", &World::set)
    ;
}
```

init<std::string>() が、std::string を引数にとるコンストラクタをエクスポートする（Python ではコンストラクタを「"__init__"」と書く）。

def()メンバ関数に init<...>を渡すことでエクスポートするコンストラクタを追加できる。例えば World に double を 2 つとる別のコンストラクタがあるとすれば、

```
class_<World>("World", init<std::string>())
    .def(init<double, double>())
    .def("greet", &World::greet)
    .def("set", &World::set)
;
```

逆にコンストラクタを 1 つもエクスポートしたくない場合は、代わりに no_init を使う。

```
class_<Abstract>("Abstract", no_init)
```

これは実際には、常に Python の RuntimeError 例外を投げる__init__メソッドを追加する。

クラスデータメンバ

データメンバもまた Python へエクスポートでき、対応する Python クラスの属性としてアクセス可能になる。各データメンバは**読み取り専用**か**読み書き可能**として見なすことができる。以下の Var クラスを考えよう。

```
struct Var
{
    Var(std::string name) : name(name), value() {}
    std::string const name;
    float value;
};
```

C++クラス Var とそのデータメンバは次のようにして Python へエクスポートできる。

```
class_<Var>("Var", init<std::string>())
    .def_readonly("name", &Var::name)
    .def_readwrite("value", &Var::value);
```

これで Python 側で hello 名前空間内に Var クラスがあるように扱うことができる。

```
>>> x = hello.Var('pi')
>>> x.value = 3.14
>>> print x.name, 'is around', x.value
pi is around 3.14
```

name を**読み取り専用**としてエクスポートしたいっぽうで、value は**読み書き可能**としてエクスポートしたことに注意していただきたい。

```
>>> x.name = 'e' # name は変更できない
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: can't set attribute
```

クラスプロパティ

C++では、公開データメンバを持つクラスは受け入れられない。カプセル化を利用して適切に設計されたクラスは、クラスのデータメンバを隠蔽しているものである。クラスのデータにアクセスする唯一の方法はアクセス関数（getter および setter）を介したものである。アクセス関数はクラスのプロパティをエクスポートする。以下がその例である。

```
struct Num
{
    Num();
    float get() const;
    void set(float value);
    ...
};
```

しかしながら Python における属性アクセスは優れたものである。ユーザが属性を直接処理しても、必ずしもカプセル化が破壊されるわけではない。属性はメソッド呼び出しの別の構文だからである。Num クラスを Boost.Python を使ってラップすると次のようになる。

```
class_<Num>("Num")
    .add_property("rovalue", &Num::get)
    .add_property("value", &Num::get, &Num::set);
```

これで Python 側は以下のようになる。

```
>>> x = Num()
>>> x.value = 3.14
>>> x.value, x.rovalue
(3.14, 3.14)
>>> x.rovalue = 2.17 # エラー！
```

以下のように rovalue の setter メンバ関数を渡していないため、クラスのプロパティ rovalue は**読み取り専用**としてエクスポートされることに注意していただきたい。

```
.add_property("rovalue", &Num::get)
```

継承

これまでの例では多態的でないクラスを扱ってきたが、通常、そうしたことはあまりない。多くの場合、多態的なクラスや継承が絡んだクラス階層をラップすることになるだろう。仮想基底クラスから派生したクラスについて Boost.Python ラッパを書かなければならなくなるだろう。

次のような簡単な継承構造を考えよう。

```
struct Base { virtual ~Base(); };
struct Derived : Base {};
```

`Base` と `Derived` インスタンスを処理する C++関数群もあるとする。

```
void b(Base*);
void d(Derived*);
Base* factory() { return new Derived; }
```

基底クラス `Base` をラップする方法は以前見た。

```
class_<Base>("Base")
    /*...*/
    ;
```

`Derived` とその基底クラスである `Base` の関係について Boost.Python に伝える。

```
class_<Derived, bases<Base> >("Derived")
    /*...*/
    ;
```

これで自動的に以下の効果が得られる:

1. `Derived` は `Base` のすべてのメソッド（ラップされた C++メンバ関数）を自動的に継承する。
2. `Base` が多態的で**ある場合**、`Base` へのポインタか参照で Python へ渡した `Derived` オブジェクトは、`Derived` へのポインタか参照が期待されているところに渡すことができる。

次に C++自由関数 `b`、`d` および `factory` をエクスポートする。

```
def("b", b);
def("d", d);
def("factory", factory);
```

自由関数 `factory` が、`Derived` クラスの新しいインスタンスを生成するために使われることに注意していただきたい。このような場合は `return_value_policy<manage_new_object>`を使って、`Base` へのポインタを受け入れ、Python のオブジェクトが破壊さ

れるまでインスタンスを新しい Python の Base オブジェクトに保持しておくことを Python に伝える。Boost.Python の呼び出しポリシーについては後で詳しく述べる。

```
// factory の結果について所有権を取るよう Python に伝える
def("factory", factory,
    return_value_policy<manage_new_object>());
```

仮想関数

本節では仮想関数を使って関数に多態的な振る舞いをさせる方法について学ぶ。前の例に引き続き、Base クラスに仮想関数を 1 つ追加しよう。

```
struct Base
{
    virtual ~Base() {}
    virtual int f() = 0;
};
```

Boost.Python の目標の 1 つが、既存の C++の設計に対して侵入を最小限にすることである。原則的にはサードパーティ製ライブラリに対して、インターフェイス部分を変更することなくエクスポート可能であるべきである。Base クラスに何かを追加するのは望ましいことではない。しかし Python 側でオーバーライドし **C++から**多態的に呼び出す関数の場合、正しく動作させるのに足場が必要になる。Python のオーバーライドが呼び出されるように仮想関数に非侵入的にフックする、Base から派生したラッパクラスを書くことである。

```
struct BaseWrap : Base, wrapper<Base>
{
    int f()
    {
        return this->get_override("f")();
    }
};
```

Base の継承に加え、wrapper<Base>を多重継承していることに注意していただきたい（ラッパの節を見よ）。wrapper テンプレートはラップするクラスを Python 側でオーバーライドできるようにする段取りを容易にする。

msvc6/7 におけるバグの回避方法

Microsoft Visual C++のバージョン 6 か 7 を使っている場合、f は次のように書かなければならない。

```
return call<int>(this->get_override("f").ptr());
```

BaseWrap のオーバーライドされた仮想メンバ関数 f は、実際には get_override を介して Python オブジェクトの相当するメソッドを呼び出す。

最後に Base をエクスポートする。

```
class_<BaseWrap, boost::noncopyable>("Base")
    .def("f", pure_virtual(&Base::f))
    ;
```

pure_virtual は、関数 f が純粋仮想関数であることを Boost.Python に伝える。

> **注意**
>
> **メンバ関数とメソッド**
>
> Python をはじめ、多くのオブジェクト指向言語では**メソッド（methods）**という用語を使う。メソッドは大雑把に言えば C++の**メンバ関数（member functions）**に相当する。

既定の実装をもった仮想関数

前節で Boost.Python のクラスラッパ機能を用いて純粋仮想関数を持ったクラスをラップする方法を見てきた。**非**純粋仮想関数をラップする場合、方法は少し異なる。

前節を思い出そう。C++で実装するか Python で派生クラスを作成する、純粋仮想関数を持ったクラスをラップした。基底クラスは次のように純粋仮想関数 f を持っていた。

```
struct Base
{
    virtual int f() = 0;
};
```

しかしながら、仮にメンバ関数 f が純粋仮想関数として宣言されていなかったら、

```
struct Base
{
    virtual ~Base() {}
    virtual int f() { return 0; }
};
```

以下のようにラップする。

```
struct BaseWrap : Base, wrapper<Base>
{
    int f()
    {
        if (override f = this->get_override("f"))
            return f(); // *注意*
        return Base::f();
    }

    int default_f() { return this->Base::f(); }
};
```

BaseWrap::f の実装方法に注意していただきたい。この場合、f のオーバーライドが存在するかチェックしなければならない。存

在しなければ Base::f() を呼び出すとよい。

MSVC6/7 におけるバグの回避方法

Microsoft Visual C++のバージョン 6 か 7 を使っている場合、＊注意＊と書いた行を次のように変更しなければならない。

```
return call<char const*>(f.ptr());
```

最後にエスクポートを行う。

```
class_<BaseWrap, boost::noncopyable>("Base")
    .def("f", &Base::f, &BaseWrap::default_f)
    ;
```

&Base::f と&BaseWrap::default_f の両方をスクスポートしていることに注意していただきたい。Boost.Python は（1）転送（dispatch）関数 f と（2）既定の実装への転送（forwarding）関数 default_f の追跡を維持しなければならない。この目的のための特別な def 関数が用意されている。

Python 側では結果的に次のようになる。

```
>>> base = Base()
>>> class Derived(Base):
...     def f(self):
...         return 42
...
>>> derived = Derived()
```

base.f() を呼び出すと次のようになる。

```
>>> base.f()
0
```

derived.f() を呼び出すと次のようになる。

```
>>> derived.f()
42
```

演算子・特殊関数

Python の演算子

C は演算子が豊富なことでよく知られている。C++はこれを演算子の多重定義を認めることにより極限まで拡張した。Boost.Python はこれを利用して、演算子を多用した C++クラスのラップを容易にする。

ファイルの位置を表すクラス FilePos と、FilePos インスタンスをとる演算子群を考える。

```
class FilePos { /*...*/ };

FilePos     operator+(FilePos, int);
FilePos     operator+(int, FilePos);
int         operator-(FilePos, FilePos);
FilePos     operator-(FilePos, int);
FilePos&    operator+=(FilePos&, int);
FilePos&    operator-=(FilePos&, int);
bool        operator<(FilePos, FilePos);
```

これらのクラスと演算子群は幾分簡単かつ直感的に Python へエクスポートできる。

```
class_<FilePos>("FilePos")
    .def(self + int())          // __add__
    .def(int() + self)          // __radd__
    .def(self - self)           // __sub__
    .def(self - int())          // __sub__
    .def(self += int())         // __iadd__
    .def(self -= other<int>())
    .def(self < self);          // __lt__
```

上記のコード片は非常に明確であり、ほとんど説明不要である。演算子のシグニチャと実質同じである。`self` が `FilePos` オブジェクトを表すということにのみ注意していただきたい。また、演算子式に現れるクラス `T` がすべて（容易に）デフォルトコンストラクト可能であるとは限らない。「self 式」を書くときに実際の `T` インスタンスの代わりに `other<T>()` が使える。

特殊メソッド

Python には他にいくつか**特殊メソッド**がある。Boost.Python は、実際の Python クラスインスタンスがサポートする標準的な特殊メソッド名をすべてサポートする。直感的なインターフェイス群で、これらの Python **特殊関数**に相当する C++関数をラップできる。以下に例を示す。

```
class Rational
{ public: operator double() const; };

Rational pow(Rational, Rational);
Rational abs(Rational);
ostream& operator<<(ostream&,Rational);

class_<Rational>("Rational")
    .def(float_(self))                  // __float__
    .def(pow(self, other<Rational>))    // __pow__
    .def(abs(self))                     // __abs__
    .def(str(self))                     // __str__
    ;
```

他に言うことは？

注意

`operator<<`の役割は？ メソッド `str` が動作するために `operator<<`が必要なのだ（`operator<<`は `def(str(self))` が定義す

るメソッドが使用する）。

関数

本章では、Boost.Python の強力な関数について詳細を見る。懸垂ポインタや懸垂参照のような潜在的な落とし穴を避けつつ、C++関数を Python へエクスポートするための機能について見ていく。また、多重定義や既定の引数といった C++機能を利用した C++関数のエクスポートを容易にする機能についても見ていく。

先を続けよう。

しかしその前に Python 2.2 以降を立ち上げて`>>> import this`とタイプしたくなるかもしれない。

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

呼び出しポリシー

C++では引数や戻り値の型としてポインタや参照を扱うことがよくある。これら単純型は非常に低水準であり表現力に乏しい。少なくとも、ポインタや参照先のオブジェクトの所有権がどこにあるか知る方法はない。もっとも、Java や Python といった言語ではそのような低水準な実体を扱うことはない。C++では、所有権のセマンティクスを正確に記述するスマートポインタの使用をよい慣習であると考えることが多い。それでも生の参照やポインタを使う C++インターフェイスがよいとされる場合もあり、Boost.Python がそれらに対処できなければならない。このためには、あなたの助けが必要である。次のような C++関数を考える。

```
X& f(Y& y, Z* z);
```

ライブラリはこの関数をどのようにラップすべきだろうか？ 単純なアプローチとしては、返される参照について Python の x オブジェクトを構築することである。この解法は動作する場合もあるが、動作しないこともある。以下が後者の例である。

```
>>> x = f(y, z) # x はC++のXを参照する
>>> del y
>>> x.some_method() # クラッシュ！
```

何が起きたのか？

実は f() が次のように実装されていたのだった。

```
X& f(Y& y, Z* z)
{
    y.z = z;
    return y.x;
}
```

問題は、f() がオブジェクト y のメンバへの参照を返すため、結果の X&の寿命が y の寿命に縛られることである。このイディオムは珍しいものではなく、C++の文脈では完全に受け入れられるものである。しかしながら Python のユーザとしてはこの C++インターフェイスを使用するだけでシステムをクラッシュさせるわけにはいかない。今回の場合、y を削除した段階で X への参照が無効となり、懸垂参照が残るのである。

以下のようなことが起こっている。

1. h への参照と z へのポインタを渡して f が呼び出される
2. y.x への参照が返される
3. y が削除される。x は懸垂参照となる
4. x.some_method() が呼び出される
5. **バン！**

結果を新しいオブジェクトにコピーしてみる。

```
>>> f(y, z).set(42) # 結果を消失
>>> y.x.get()       # クラッシュしないが、改善の余地がある
3.14
```

これは今回の C++インターフェイスで本当に実現したかったことではない。Python インターフェイスが可能な限り綿密に C++インターフェイスを反映すべきであるという約束を破っている。

問題はこれで終わりではない。Y の実装が次のようになっているとしたら、

```
struct Y
{
    X x; Z* z;
    int z_value() { return z->value(); }
};
```

データメンバ z がクラス Y に生のポインタで保持されていることに注意していただきたい。潜在的な懸垂ポインタの問題が Y の内部で発生している。

```
>>> x = f(y, z) # y は z を参照する
>>> del z       # オブジェクト z を削除
>>> y.z_value() # クラッシュ！
```

参考のために f の実装を再掲する。

```
X& f(Y& y, Z* z)
{
    y.z = z;
    return y.x;
}
```

以下のようなことが起こっている。

1. y への参照と z へのポインタを渡して f が呼び出される
2. y が z へのポインタを保持する
3. y.x への参照が返される
4. z が削除される。y.z は懸垂ポインタとなる
5. y.z_value() が呼び出される
6. z->value() が呼び出される
7. **バン！**

呼び出しポリシー

上で扱った例のような状況では、呼び出しポリシーが使える。今回の例では return_internal_reference と with_custodian_and_ward が助けになる。

```
def("f", f,
    return_internal_reference<1,
        with_custodian_and_ward<1, 2> >());
```

引数の 1 とか 2 って何だい？

```
return_internal_reference<1
```

これは 1 番目の引数（Y& y）が返される参照（X&）の所有者であると Boost.Python に伝えている。「1」は単に 1 番目の引数という意味である。まとめると「第 1 引数 Y& y が所有する内部参照 X&を返す」となる。

```
with_custodian_and_ward<1, 2>
```

これは ward（被後見人）で指定した引数（第 2 引数。Z* z）の寿命が custodian（後見人）で指定した引数（第 1 引数。Y& y）の寿命に依存すると Boost.Python に伝えている。

上で 2 つのポリシーを定義していることに注意していただきたい。2 つ以上のポリシーは数珠繋ぎに結合できる。汎用的な構文は以下のようになる。

```
policy1<args...,
    policy2<args...,
        policy3<args...> > >
```

定義済みの呼び出しポリシーを以下のリストに挙げる。完全なリファレンスはここにある。

- **with_custodian_and_ward**：引数の寿命を他の引数で縛る
- **with_custodian_and_ward_postcall**：引数の寿命を他の引数や返り値で縛る
- **return_internal_reference**：1 つの引数の寿命を返り値の寿命で縛る
- **return_value_policy<T>（T は以下のいずれか）：**
  - **reference_existing_object**：単純（で危険）なアプローチ
  - **copy_const_reference**：Boost.Python v1 のアプローチ
  - **copy_non_const_reference**：
  - **manage_new_object**：ポインタを受け取りインスタンスを保持する

禅（Zen）を思い出そう、Luke：[2]

「ごちゃごちゃ難しいのより、白黒はっきりしてるのがいい」

「あいまいなことをてきとーに処理しちゃいけません」

多重定義

多重定義したメンバ関数を手動でラップする方法を以下に示す。非メンバ関数の多重定義をラップする場合も、当然同様のテクニックが使える。

次のような C++クラスを考える。

```
struct X
{
    bool f(int a)
    {
        return true;
    }

    bool f(int a, double b)
    {
        return true;
    }

    bool f(int a, double b, char c)
    {
        return true;
    }
```

2　訳注　この日本語訳は http://www.python.jp/Zope/Zope/articles/misc/zen によりました（Copyright © 2001-2012 Python Japan User's Group）。

```
    int f(int a, int b, int c)
    {
        return a + b + c;
    };
};
```

クラス x に多重定義された関数が 4 つある。まずメンバ関数ポインタ変数を導入するところから始める。

```
bool    (X::*fx1)(int)              = &X::f;
bool    (X::*fx2)(int, double)      = &X::f;
bool    (X::*fx3)(int, double, char)= &X::f;
int     (X::*fx4)(int, int, int)    = &X::f;
```

これがあれば、続けて Python のために定義とラップができる。

```
.def("f", fx1)
.def("f", fx2)
.def("f", fx3)
.def("f", fx4)
```

既定の引数

Boost.Python は（メンバ）関数ポインタをラップするが、残念ながら C++関数ポインタは既定の引数について情報を持たない。既定の引数を持った関数 f を考える。

```
int f(int, double = 3.14, char const* = "hello");
```

しかし関数 f へのポインタ型は、その既定の引数について情報を持たない。

```
int(*g)(int,double,char const*) = f;     // 既定の引数が失われる！
```

この関数ポインタを def 関数へ渡すとしても、既定の引数を取得する方法はない。

```
def("f", f);                             // 既定の引数が失われる！
```

このため C++ラップコードを書くときは、前節で示したような手動のラップか薄いラッパを書くことに頼るしかない。

```
// 「薄いラッパ」を書く
int f1(int x) { return f(x); }
int f2(int x, double y) { return f(x,y); }

/*...*/

    // init モジュール内
    def("f", f);  // 3 引数バージョン
    def("f", f2); // 2 引数バージョン
```

```
    def("f", f1); // 1引数バージョン
```

以下のいずれかの関数（メンバ関数）をラップするときは、次節に進むとよい。

- 既定の引数を持つ
- 引数の先頭部分に共通列を持つ形で多重定義されている

BOOST_PYTHON_FUNCTION_OVERLOADS

Boost.Python はこれを容易にする方法を提供する。例えば次の関数が与えられたとする。

```
int foo(int a, char b = 1, unsigned c = 2, double d = 3)
{
    /*...*/
}
```

次のマクロ呼び出しにより、薄いラッパが作成される。

```
BOOST_PYTHON_FUNCTION_OVERLOADS(foo_overloads, foo, 1, 4)
```

このマクロは、`def(...)`に渡すことができる`foo_overloads`クラスを作成する。このマクロの 3 番目と 4 番目の引数は、それぞれ引数の最小数と最大数である。`foo`関数では引数の最小数は 1、最大数は 4 である。`def(...)`関数は`foo`のファミリをすべて自動的に追加する。

```
def("foo", foo, foo_overloads());
```

BOOST_PYTHON_MEMBER_FUNCTION_OVERLOADS

オブジェクトはここにも、そこにも、あそこにも、どこにでもある。Python にエクスポートするのは、クラスのメンバ関数が最も頻度が高い。ここでまた、以前の既定の引数や引数の先頭部分が共通列である多重定義の場合の不便が出てくる。これを容易にするマクロが提供されている。

`BOOST_PYTHON_FUNCTION_OVERLOADS`と同様、メンバ関数をラップする薄いラッパを自動的に作成するのに`BOOST_PYTHON_MEMBER_FUNCTION_OVERLOADS`を使用する。例を挙げる。

```
struct george
{
    void
    wack_em(int a, int b = 0, char c = 'x')
    {
        /*...*/
    }
};
```

ここで次のようにマクロを呼び出すと、

```
BOOST_PYTHON_MEMBER_FUNCTION_OVERLOADS(george_overloads, wack_em, 1, 3)
```

george の wack_em メンバ関数について最少で 1、最多で 3（マクロの 3 番目と 4 番目の引数）の薄いラッパ群を生成する。薄いラッパはすべて george_overloads という名前のクラスに収められ、def(...)に引数として渡すことができる。

```
.def("wack_em", &george::wack_em, george_overloads());
```

詳細は多重定義のリファレンスを見よ。

init と optional

クラスのコンストラクタ、特に既定の引数と多重定義については類似の機能が提供されている。init<...>を覚えているだろうか？ 例えばクラス X とそのコンストラクタがあるとすると、

```
struct X
{
    X(int a, char b = 'D', std::string c = "constructor", double d = 0.0);
    /*...*/
}
```

このコンストラクタを 1 発で Boost.Python に追加するには、

```
.def(init<int, optional<char, std::string, double> >())
```

init<...>と optional<...>の使用が既定（省略可能な引数）を表すことに注意していただきたい。

自動多重定義

前節で BOOST_PYTHON_FUNCTION_OVERLOADS および BOOST_PYTHON_MEMBER_FUNCTION_OVERLOADS が、引数列の先頭部分が共通である多重定義関数およびメンバ関数に対しても使用できることを見た。以下に例を示す。

```
void foo()
{
   /*...*/
}

void foo(bool a)
{
   /*...*/
}

void foo(bool a, int b)
{
   /*...*/
}

void foo(bool a, int b, char c)
{
```

```
    /*...*/
}
```

前節と同様、これらの多重定義された関数について薄いラッパを一発で生成できる。

```
BOOST_PYTHON_FUNCTION_OVERLOADS(foo_overloads, foo, 0, 3)
```

その結果、次のように書ける。

```
.def("foo", (void(*)(bool, int, char))0, foo_overloads());
```

この例では引数の個数は最少で 0、最多で 3 となっていることに注意していただきたい。

手動のラッピング

多重定義した関数は引数列の先頭に共通部分を持っていなければならないということを強調しておく。それ以外の場合、上で述べた方法は動作せず、関数を手動でラップしなければならない。

実際には多重定義関数の手動ラッピングと、`BOOST_PYTHON_MEMBER_FUNCTION_OVERLOADS` とその姉妹版である `BOOST_PYTHON_FUNCTION_OVERLOADS` による自動的なラッピングを混用することは可能である。多重定義の節で見た例だと 4 つの多重定義関数は引数の先頭列が共通であるので、`BOOST_PYTHON_MEMBER_FUNCTION_OVERLOADS` を使って最初の 3 つの `def` を自動的にラップでき、残り 1 つだけを手動でラップすることになる。方法は以下のとおり。

```
BOOST_PYTHON_MEMBER_FUNCTION_OVERLOADS(xf_overloads, f, 1, 4)
```

両方の `X::f` 多重定義について、メンバ関数ポインタを作成すると、

```
bool    (X::*fx1)(int, double, char)    = &X::f;
int     (X::*fx2)(int, int, int)        = &X::f;
```

結果、以下のように書ける。

```
.def("f", fx1, xf_overloads());
.def("f", fx2)
```

オブジェクトのインターフェイス

C++が静的型付けであるのに対し、Python は動的型付けである。Python の変数は整数、浮動小数点数、リスト、辞書、タプル、文字列、長整数、その他を保持できる。Boost.Python と C++の視点では、これら Python 的な変数は `object` クラスのインスタンスにすぎない。本章で Python のオブジェクトをどのように扱うか見ていく。

以前述べたように Boost.Python の目的の 1 つは、C++と Python 間における Python 的な感覚の双方向マッピングの提供である。Boost.Python における C++の `object` は可能な限り Python に類似したものとなっている。これにより学習曲線は著しく最小化される

はずである。

基本的なインターフェイス

`object` クラスは `PyObject*`をラップする。参照カウントの管理といった `PyObject` の複雑な取り扱いは、すべて `object` クラスが処理する。C++オブジェクトの相互運用性はシームレスなものである。実際のところ、Boost.Python における C++の `object` はあらゆる C++オブジェクトから明示的に構築できる。

説明のために、以下のような Python コード片を考える。

```
def f(x, y):
     if (y == 'foo'):
          x[3:7] = 'bar'
     else:
          x.items += y(3, x)
     return x

def getfunc():
   return f;
```

Boost.Python の機能を用いて C++で書き直すと次のようになる。

```
object f(object x, object y) {
     if (y == "foo")
          x.slice(3,7) = "bar";
     else
          x.attr("items") += y(3, x);
     return x;
}
object getfunc() {
    return object(f);
}
```

C++でコードを書いているという外観的な差を除けば、そのルックアンドフィールは Python のプログラマにも明確である。

object の派生型

Boost.Python には、Python の各型に対応する `object` の派生型がある。

- `list`
- `dict`
- `tuple`
- `str`
- `long_`
- `enum_`

これらの `object` の派生型は実際の Python 型と同様に振舞う。例を挙げる。

```
str(1) ==> "1"
```

個々の派生 object は対応する Python 型のメソッドを持つ。例えば dict は keys() メソッドを持つ。

```
d.keys()
```

タプルリテラルを宣言するのに make_tuple が提供されている。例を挙げる。

```
make_tuple(123, 'D', "Hello, World", 0.0);
```

C++において、Boost.Python の object を関数の引数に渡す場合は派生型の一致が要求される。例えば以下に示す関数 f をラップする場合、Python の str 型とその派生型のみを受け付ける。

```
void f(str name)
{
    object n2 = name.attr("upper")();    // NAME = name.upper()
    str NAME = name.upper();             // こちらのほうがよい
    object msg = "%s is bigger than %s" % make_tuple(NAME,name);
}
```

細かく見ると、

```
str NAME = name.upper();
```

このコードから分かるように、str 型のメソッドを C++メンバ関数として提供している。次に、

```
object msg = "%s is bigger than %s" % make_tuple(NAME,name);
```

上記のコードのように Python の "format" % x,y,z を C++で書ける。標準の C++で同じことを簡単に行う方法がないため便利である。

Python 同様、Python の可変型の多くがコンストラクタでコピーを行うというよく知られた落とし穴があるので注意が必要である。

Python の場合：

```
>>> d = dict(x.__dict__)      # x.__dict__をコピーする
>>> d['whatever'] = 3         # コピーを変更する
```

C++の場合：

```
dict d(x.attr("__dict__"));  // x.__dict__をコピーする
d['whatever'] = 3;           // コピーを変更する
```

object としての class_<T>

Boost.Python における object の動的な性質に従えば、あらゆる class_<T>もまたこれら型の 1 つである！ 以下のコード片はクラス（型）オブジェクトをラップする。

これを使って、ラップされたインスタンスを作成できる。

```
object vec345 = (
    class_<Vec2>("Vec2", init<double, double>())
        .def_readonly("length", &Point::length)
        .def_readonly("angle", &Point::angle)
    )(3.0, 4.0);

assert(vec345.attr("length") == 5.0);
```

C++オブジェクトの抽出

object インスタンスを使用せずに C++の値が必要になることがある。これは extract<T>関数で実現できる。以下を考える。

```
double x = o.attr("length"); // コンパイルエラー
```

Boost.Python の object は double へ暗黙に変換できないため、上記のコードはコンパイルエラーとなる。代わりに以下のように書けば希望どおりとなる。

```
double l = extract<double>(o.attr("length"));
Vec2& v = extract<Vec2&>(o);
assert(l == v.length());
```

1 行目は Boost.Python の object の length 属性を抽出しようとしている。2 行目は Boost.Python の object が保持している Vec2 オブジェクトを**抽出**しようとしている。

「～しようとしている」と書いたことに注意していただきたい。Boost.Python の object が実際には Vec2 型を保持していなかったらどうなるだろうか？ これは Python の object がもつ動的な性質を考えれば十分ありうることである。安全のため、希望する C++型を抽出できない場合は適当な例外が投げられる。例外を避けるには抽出できるかテストする必要がある。

```
extract<Vec2&> x(o);
if (x.check()) {
    Vec2& v = x(); ...
```

明敏な読者は extract<T>の機能が変更可能コピーの問題を解決することに気付いたかもしれない。

```
dict d = extract<dict>(x.attr("__dict__"));
d["whatever"] = 3;          // x.__dict__を変更する！
```

列挙

Boost.Python には、C++の列挙を捕捉、ラップする気の利いた機能がある。Python に `enum` 型はないが、C++の列挙を Python へ `int` としてエクスポートしたいことがよくある。Python の動的型付けから C++の強い静的型付けへの適切な変換に気を付けていれば Boost.Python の列挙機能で容易に可能である（C++では、整数から列挙へ暗黙に変換することはできない）。次のような C++の列挙があったとして、

```
enum choice { red, blue };
```

次のようにして Python へエクスポートする。

```
enum_<choice>("choice")
    .value("red", red)
    .value("blue", blue)
    ;
```

新しい列挙は現在の `scope()` に作成される。これは大抵の場合現在のモジュールである。上記のコード片は Python の `int` 型から派生した、第 1 引数に渡した C++型に対応する Python クラスを作成する。

注意

スコープとは

スコープは、新しい拡張クラスやラップした関数が属性として定義される Python の名前空間を制御するグローバルな関連 Python オブジェクトを持つクラスである。詳細はリファレンスを見よ。

Python からはこれらの値に以下のようにしてアクセスできる。

```
>>> my_module.choice.red
my_module.choice.red
```

ここで `my_module` は列挙を宣言したモジュールである。新しいスコープをクラスに対して作成することもできる。

```
scope in_X = class_<X>("X")
                .def( ... )
                .def( ... )
            ;

// X::nested を X.nested としてエクスポートする
enum_<X::nested>("nested")
    .value("red", red)
    .value("blue", blue)
    ;
```

PyObject*から boost::python::object を作成する

`PyObject*`である `pyobj` へのポインタを `boost::python::object` で管理したい場合、以下のようにする。

```
boost::python::object o(boost::python::handle<>(pyobj));
```

この場合、オブジェクト o は pyobj を管理するが、構築時に参照カウントを増やさない。

あるいは借用（borrowed）参照を使う方法として、

```
boost::python::object o(boost::python::handle<>(boost::python::borrowed(pyobj)));
```

この場合 Py_INCREF が呼び出されるので、オブジェクト o がスコープ外に出ても pyobj は破壊されない。

組み込み

Boost.Python を使って Python から C++のコードを呼び出す方法について理解できたと思う。しかしときには逆のこと、つまり C++側から Python のコードを呼び出す必要が出てくるはずである。これには Python のインタープリタを C++のプログラムに**組み込む**必要がある。

現時点では Boost.Python は組み込みに必要なことをすべてサポートしているわけではない。したがってこのギャップを埋めるには Python の C API を使う必要が出てくる。とはいえ Boost.Python は組み込みの大部分を容易にしており、将来のバージョンでは Python の C API に触れる必要はなくなるかもしれない。そういうわけだから期待しておいて欲しい。

組み込みプログラムをビルドする

Python をプログラムに組み込み可能にするには、Python だけでなく Boost.Python 本体の実行時ライブラリにもリンクしなければならない。

Boost.Python のライブラリは 2 種類ある。いずれも Boost の /libs/python/build/bin-stage サブディレクトリにある。Windows ではライブラリの名前は boost_python.lib（リリースビルド用）と boost_python_debug.lib（デバッグ用）である。ライブラリが見つからない場合は、おそらくまだ Boost.Python をビルドしていないのだろう。ビルドとテストを見て方法を確認するとよい。

Python のライブラリは、Python ディレクトリの /libs サブディレクトリにある。Windows では pythonXY.lib のような名前で、X.Y が Python のメジャーバージョンの番号である。

また Python の /include サブディレクトリをインクルードパスに追加しておかなければならない。

Jamfile に以上のことをすべて要約すると、

```
projectroot c:\projects\embedded_program ; # プログラムの場所

# Python 用の規則
SEARCH on python.jam = $(BOOST_BUILD_PATH) ;
include python.jam ;

exe embedded_program # 実行可能ファイルの名前
  : # ソースファイル
    embedded_program.cpp
  : # 必須条件
    <find-library>boost_python <library-path>c:\boost\libs\python
  $(PYTHON_PROPERTIES)
```

```
    <library-path>$(PYTHON_LIB_PATH)
    <find-library>$(PYTHON_EMBEDDED_LIBRARY) ;
```

はじめに

ビルドできるようになったのはよいが、まだビルドするものがない。Python のインタープリタを C++のプログラムに組み込むには、以下の 3 段階が必要である。

1. #include <boost/python.hpp>
2. Py_Initialize()を呼び出してインタープリタを起動、__main__モジュールを作成する。
3. 他の Python C API を呼び出してインタープリタを使用する。

> **注意**
>
> **現時点ではインタープリタを停止するのに Py_Finalize()を呼び出してはならない。これは Boost.Python の将来のバージョンで修正する。**

（当然ながら、上記の段階の間に C++コードが入る。）

これでプログラムにインタープリタを組み込み可能になった。次に使用方法を見ていく。

インタープリタを使用する

すでに知っていることと思うが、Python のオブジェクトは参照カウントで管理されている。当然、Python C API の PyObject も参照カウンタを持っているが、違いがある。参照カウントは Python では完全に自動で行われているが、Python C API では手動で行う必要がある。これは厄介で、とりわけ C++例外が現れるコードで正しく取り扱うのが困難である。幸いにも Boost.Python には handle および object クラステンプレートがあり、この処理を自動化できる。

Python のコードを起動する

Boost.Python は、C++から Python のコードを起動する関数を 3 つ提供している。

```
object eval(str expression, object globals = object(), object locals = object())
object exec(str code, object globals = object(), object locals = object())
object exec_file(str filename, object globals = object(), object locals = object())
```

eval は与えられた式を評価し結果の値を返す。exec は与えられたコード（典型的には文の集まり）を実行し結果を返す。exec_file は与えられたファイル内のコードを実行する。

globals と locals 引数は、コードを実行するコンテキストの globals と locals に相当する Python の辞書である。ほとんどの目的において、__main__モジュールの名前空間辞書を両方の引数に使用するとよい。

Boost.Python はモジュールをインポートする関数を提供する。

```
object import(str name)
```

`import` は Python のモジュールをインポートし（潜在的には、はじめに起動しているプロセスに読み込む）、返す。

`__main__`モジュールをインポートし、その名前空間で Python のコードを走らせてみよう。

```
object main_module = import("__main__");
object main_namespace = main_module.attr("__dict__");

object ignored = exec("hello = file('hello.txt', 'w')\n"
                      "hello.write('Hello world!')\n"
                      "hello.close()",
                      main_namespace);
```

このコードは現在のディレクトリに `hello.txt` という名前のファイルを作成し、プログラミングサークルでよく知られたフレーズを書き込む。

Python のオブジェクトを操作する

Python オブジェクトを操作するクラスを用意したいことがよくある。しかしすでに上記や前節でそのようなクラスを見た。文字通りの名前を持つ `object` とその派生型である。またそれらを `handle` から構築できることも見た。以下の例を見ればより明らかだろう。

```
object main_module = import("__main__");
object main_namespace = main_module.attr("__dict__");
object ignored = exec("result = 5 ** 2", main_namespace);
int five_squared = extract<int>(main_namespace["result"]);
```

`__main__`モジュールの名前空間に相当する辞書オブジェクトを作成している。次に 5 の 2 乗を結果の変数に代入し、この変数を辞書から読んでいる。同じ結果を得る他の方法としては代わりに `eval` を使用する方法があり、こちらは結果を直接返す。

```
object result = eval("5 ** 2");
int five_squared = extract<int>(result);
```

例外処理

Python の式を評価中に例外を送出した場合、`error_already_set` が投げられる。

```
try
{
    object result = eval("5/0");
    // ここには絶対に来ない：
    int five_divided_by_zero = extract<int>(result);
}
catch(error_already_set const &)
{
    // 何らかの方法で例外を処理する
}
```

error_already_set 例外クラス自体は何の情報も持たない。送出された Python の例外について詳細を調べるには、catch 文内で Python C API の例外処理関数を使用する必要がある。これは単純に PyErr_Print() を呼び出して例外のトレースバックをコンソールへプリントするか、あるいは例外の型を標準の例外と比較する程度となるだろう。

```
catch(error_already_set const &)
{
    if (PyErr_ExceptionMatches(PyExc_ZeroDivisionError))
    {
        // ZeroDivisionError を個別に処理する
    }
    else
    {
        // 他のすべてのエラーを stderr にプリントする
        PyErr_Print();
    }
}
```

（例外についてより多くの情報を取得するには、このリストにある例外処理関数を使用する。）

イテレータ

C++、特に STL においてイテレータはあらゆる場面で使用されている。Python にもイテレータがあるが、両者には大きな違いがある。

C++のイテレータ:

- C++のイテレータは 5 つに分類される（ランダムアクセス、双方向、単方向、入力、出力）
- 再配置とアクセスの 2 種類の操作がある
- 範囲を表すのにイテレータの組（先頭と末尾）が必要

Python のイテレータ:

- 分類は 1 つしかない（単方向）
- 操作は 1 種類しかない（next()）
- 終了時に StopIteration 例外を投げる

典型的な Python の走査プロトコルである for y in x...は以下のようである。

```
iter = x.__iter__()          # イテレータを取得する
try:
    while 1:
    y = iter.next()          # 各要素を取得する
    ...                      # y を処理する
except StopIteration: pass   # イテレータが尽きた
```

Boost.Python は、C++のイテレータを Python のイテレータとして振舞うようにする機構をいくつか提供している。必要なことは C++のイテレータから Python の走査プロトコルと互換性のある適切な`__iter__`関数を用意することである。例えば、

```
object get_iterator = iterator<vector<int> >();
object iter = get_iterator(v);
object first = iter.next();
```

あるいは`class_<>`で以下のようにする。

```
.def("__iter__", iterator<vector<int> >())
```

range

`range`関数を使用すると、Python の実践的なイテレータを作成できる。

- `range(start, finish)`
- `range<Policies, Target>(start, finish)`

ここで *start*、*finish* は以下のいずれかである。

- メンバデータポインタ
- メンバ関数ポインタ
- 関数オブジェクト（*Target* 引数を使用）

iterator

- `iterator<T, Policies>()`

コンテナ`T`が与えられた場合、`iterator`は単に`&T::begin`と`&T::end`で`range`を呼び出すショートカットとなる。

実際にやってみよう。以下はある仮説の粒子加速器のコードからの例である。

```
f = Field()
for x in f.pions:
    smash(x)
for y in f.bogons:
    count(y)
```

C++のラッパは以下のようになるだろう。

```
class_<F>("Field")
    .property("pions", range(&F::p_begin, &F::p_end))
    .property("bogons", range(&F::b_begin, &F::b_end));
```

stl_input_iterator

ここまで C++のイテレータと範囲を Python へエクスポートする方法を見てきた。しかし時には違うこと、Python のシーケンスを STL アルゴリズムに渡したり、STL コンテナを初期化したいと考えることがある。Python のイテレータを STL のイテレータのように見せかける必要がある。これには `stl_input_iterator<>`を使用する。`std::list<int>::assign()`を Python へエクスポートする関数の実装方法を考えよう。

```
template<typename T>
void list_assign(std::list<T>& l, object o) {
    // Python のシーケンスを STL の入力範囲に変換する
    stl_input_iterator<T> begin(o), end;
    l.assign(begin, end);
}

// list<int>のラッパの一部
class_<std::list<int> >("list_int")
    .def("assign", &list_assign<int>)
    // ...
    ;
```

これで Python 側であらゆる整数シーケンスを `list_int` オブジェクトへ代入できる。

```
x = list_int();
x.assign([1,2,3,4,5])
```

例外の変換

C++の例外はすべて Python コードとの境界で捕捉しなければならない。この境界は C++が Python と接する地点である。Boost.Python は選択した標準の例外を変換してダウンする既定の例外ハンドラを提供する。

```
raise RuntimeError, 'unidentifiable C++ Exception'
```

ユーザがカスタムの変換器を提供してもよい。例えば、[3]

```
struct PodBayDoorException;
void translator(PodBayDoorException const& x) {
    PyErr_SetString(PyExc_UserWarning, "I'm sorry Dave...");
}
BOOST_PYTHON_MODULE(kubrick) {
     register_exception_translator<
          PodBayDoorException>(translator);
     ...
```

3　訳注 『2001 年宇宙の旅』(“2001: A Space Odyssey” : Stanley Kubrick and Arthur C. Clarke, 1968)かな？

典型的なテクニック

Boost.Python でコードをラップするのに使えるテクニックをいくつか紹介する。

パッケージを作成する

Python のパッケージは、ユーザに一定の機能を提供するモジュールの集まりである。パッケージの作成についてなじみがなければ、Python のチュートリアルによい導入がある。

しかし今は Boost.Python を使って C++コードをラップしているのである。優れたパッケージインターフェイスをユーザに提供するにはどうすればよいだろうか？ 概念的なことを捉えるために例を使って考えよう。

音に関する C++ライブラリがあったとする。様々な形式で読み書きし、音データにフィルタをかける等するものとする。（便宜的に）名前を `sounds` としておこう。以下のような整理された C++名前空間の階層がすでにあるとする。

```
sounds::core
sounds::io
sounds::filters
```

Python ユーザに同じ階層を提示し、次のようなコードが書けるようにしたい。

```
import sounds.filters
sounds.filters.echo(...) # echo は C++関数
```

第 1 段階はラップコードを書くことである。以下のように Boost.Python を使って各モジュールを個別にエクスポートしなければならない。

```
/* ファイル core.cpp */
BOOST_PYTHON_MODULE(core)
{
    /* 名前空間 sounds::core 内のものをすべてエクスポートする */
    ...
}

/* ファイル io.cpp */
BOOST_PYTHON_MODULE(io)
{
    /* 名前空間 sounds::io 内のものをすべてエクスポートする */
    ...
}

/* ファイル filters.cpp */
BOOST_PYTHON_MODULE(filters)
{
    /* 名前空間 sounds::filters 内のものをすべてエクスポートする */
    ...
}
```

これらのファイルをコンパイルすると、`core.pyd`、`io.pyd` および `filters.pyd` の Python 拡張が生成される。

注意

拡張子 `.pyd` は Python の拡張モジュールで使用するものであり、単純に共有ライブラリである。システムで既定のもの（Unix の場合は `.so`、Windows の場合は `.dll`）を使用しても差し支えない。

次に以下の Python パッケージ用のディレクトリ構造を作成する。

```
sounds/
    __init__.py
    core.pyd
    filters.pyd
    io.pyd
```

ファイル `__init__.py` は、ディレクトリ `sounds/` が実際は Python のパッケージであることを Python に伝える。このファイルは空でもよいが、後述するようにここでマジックを行うことも可能だ。

これでパッケージの準備が整った。ユーザがなすべきなのは、`sounds` を PYTHONPATH に置いてインタープリタを起動することだけである。

```
>>> import sounds.io
>>> import sounds.filters
>>> sound = sounds.io.open('file.mp3')
>>> new_sound = sounds.filters.echo(sound, 1.0)
```

何も問題ないようだが、どうだろう？

これはパッケージ階層を作成する最も単純な方法だが、柔軟性がまるでない。**純粋な** Python の関数、例えば音オブジェクトに 3 つのフィルタを同時にかける関数を `filters` パッケージに追加したい場合はどうだろうか？ 確かに C++で書いてエクスポートすれば可能だが、Python でやってみてはどうか。そうすれば拡張モジュールの再コンパイルが不要で、書くのも簡単である。

こういった柔軟性が必要な場合、パッケージ階層を少しばかり複雑にしなければならない。まず拡張モジュール群の名前を変更しなければならない。

```
/* ファイル core.cpp */
BOOST_PYTHON_MODULE(_core)
{
    ...
    /* 名前空間 sounds::core 内のものをすべてエクスポートする */
}
```

モジュール名に下線を追加したことに注意していただきたい。ファイル名も `_core.pyd` に変わるはずである。他の拡張モジュールも同様である。これでパッケージ階層は以下のように変更された。

```
sounds/
    __init__.py
    core/
        __init__.py
        _core.pyd
    filters/
```

```
        __init__.py
        _filters.pyd
    io/
        __init__.py
        _io.pyd
```

各拡張モジュールについてディレクトリを作成し、それぞれに__init__.pyを追加したことに注意していただきたい。しかしこれをこのままおいておくと、ユーザは次のような構文でcoreモジュールの関数にアクセスしなければならない。

```
>>> import sounds.core._core
>>> sounds.core._core.foo(...)
```

これは望ましいことではない。しかしここで__init__.pyのマジックが発動する。__init__.pyの名前空間に持ち込まれるものはすべてユーザが直接アクセスできるのである。そういうわけで、名前空間全体を_core.pydからcore/__init__.pyへ持ち込むだけでよい。つまり次のコード行をsounds/core/__init__.pyへ追加する。

```
from _core import *
```

他のパッケージも同様に行う。これでユーザは以前のように拡張モジュール内と関数とクラスにアクセスできるようになる。

```
>>> import sounds.filters
>>> sounds.filters.echo(...)
```

他にも純粋なPython関数をあらゆるモジュールに容易に追加できるという利点もある。この方法であればユーザにはC++関数とPython関数の見分けが付かない。では**純粋な**Python関数echo_noiseをfiltersパッケージに追加しよう。この関数は与えられたsoundオブジェクトにechoとnoiseの両方のフィルタを順番に適用する。sounds/filters/echo_noise.pyという名前でファイルを作成して関数のコードを書く。

```
import _filters
def echo_noise(sound):
    s = _filters.echo(sound)
    s = _filters.noise(sound)
    return s
```

次に以下の行をsounds/filters/__init__.pyに追加する。

```
from echo_noise import echo_noise
```

これで終わりだ。ユーザは、filtersパッケージの他の関数と同様にこの関数にアクセスできる。

```
>>> import sounds.filters
>>> sounds.filters.echo_noise(...)
```

ラップしたオブジェクトを Python で拡張する

Python の柔軟性に感謝することだ。クラスを作成した後であってもメソッドを容易に追加できる。

```
>>> class C(object): pass
>>>
>>> # 普通の関数
>>> def C_str(self): return 'C のインスタンス！'
>>>
>>> # メンバ関数に変更する
>>> C.__str__ = C_str
>>>
>>> c = C()
>>> print c
C のインスタンス！
>>> C_str(c)
C のインスタンス！
```

やはり Python は素晴らしい。

同様のことが Boost.Python でラップしたクラスでもできる。C++側に `point` クラスがあるとする。

```
class point {...};

BOOST_PYTHON_MODULE(_geom)
{
    class_<point>("point")...;
}
```

前節『パッケージを作成する』のテクニックを使うと `geom/__init__.py` に直接コードが書ける。

```
from _geom import *

# 普通の関数
def point_str(self):
    return str((self.x, self.y))

# メンバ関数に変更する
point.__str__ = point_str
```

C++で作成した**すべての** `point` インスタンスがこのメンバ関数を持つことになる！ このテクニックには色々と利点がある。

- 追加する関数についてのコンパイル時間増加がゼロになる
- メモリのフットプリントが見かけ上ゼロに削減する
- 再コンパイルの必要が最小になる
- 高速なプロトタイピング（インターフェイスを変更しないことが要求されている場合、コードを C++に移動することが可能）

メタクラスを使って簡単な構文糖を追加することもできる。メソッドを他のクラスに「注入する」特別なメタクラスを作成しよう。

```
# Boost.Python がすべてのラップされたクラスに対して使用するもの。
# "point" の代わりに Boost でエクスポートしたあらゆるクラスが使用できる
BoostPythonMetaclass = point.__class__

class injector(object):
    class __metaclass__(BoostPythonMetaclass):
        def __init__(self, name, bases, dict):
            for b in bases:
                if type(b) not in (self, type):
                    for k,v in dict.items():
                        setattr(b,k,v)
            return type.__init__(self, name, bases, dict)

# point にいくつかメソッドを注入する
class more_point(injector, point):
    def __repr__(self):
        return 'Point(x=%s, y=%s)' % (self.x, self.y)
    def foo(self):
        print 'foo!'
```

これでどうなるか見てみよう。

```
>>> print point()
Point(x=10, y=10)
>>> point().foo()
foo!
```

別の有用な考えとして、コンストラクタをファクトリ関数で置き換える方法がある。

```
_point = point

def point(x=0, y=0):
    return _point(x, y)
```

このような簡単な例ではつまらない感じがするが、多重定義や引数が多数あるコンストラクタにおいては優れた単純化となることが多い。キーワードサポートに対してコンパイル時間のオーバーヘッドがゼロ、メモリのフットプリントも事実上ゼロとなる。

コンパイルにかかる時間を短縮する

多数のクラスをエクスポートすると、Boost.Python ラッパのコンパイルにかなりの時間がかかる。またメモリの消費量が容易に過大となる。これが問題となるのであれば、class_定義を複数のファイルに分割するとよい。

```
/* ファイル point.cpp */
#include <point.h>
#include <boost/python.hpp>

void export_point()
{
    class_<point>("point")...;
}

/* ファイル triangle.cpp */
```

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

void export_triangle()
{
    class_<triangle>("triangle")...;
}
```

そして BOOST_PYTHON_MODULE マクロを含んだ main.cpp ファイルを作成し、その中でエクスポート関数を呼び出す。

```
void export_point();
void export_triangle();

BOOST_PYTHON_MODULE(_geom)
{
    export_point();
    export_triangle();
}
```

これらのファイルをすべてコンパイル、リンクすると、通常の方法の場合と同じ結果が得られる。しかしメモリはまともな状態が維持できる。

```
#include <boost/python.hpp>
#include <point.h>
#include <triangle.h>

BOOST_PYTHON_MODULE(_geom)
{
    class_<point>("point")...;
    class_<triangle>("triangle")...;
}
```

C++ライブラリ開発と Python へのエクスポートを同時に行っている場合にも、この方法を推奨する。クラス内で変更があっても、ラッパコード全体ではなく単一の cpp ファイルについてコンパイルが必要になるだけである。

注意

Pyste を使ってクラスをエクスポートする場合は、--multiple オプションを覚えておくとよい。ここで示したように複数のファイルにラッパを生成する。

注意

巨大なソースファイルをコンパイルしてエラーメッセージ「致命的なエラー C1204:コンパイラの制限:内部構造がオーバーフローしました。」が出た場合にも、この方法を推奨する。FAQ に説明がある。

ビルトとテスト

1 必要事項

Boost.Python は Python のバージョン 2.2[4]かそれ以降を要求する。

2 背景

C++と Python を組み合わせるための基本的なモデルは 2 つある。

- 拡張。エンドユーザは Python のインタープリタ（実行可能ファイル）を起動し、C++で書かれた Python の「拡張モジュール」をインポートする。ライブラリを C++で書き Python のインターフェイスを与えることで、Python のプログラマが使用できるようにするという考え。Python からはこれらのライブラリは通常の Python モジュールと同じにしか見えない。
- 組み込み。エンドユーザは、Python をライブラリのサブルーチンとして呼び出す C++で書かれたプログラムを起動する。既存のアプリケーションにスクリプト機能を追加するという考え。

拡張と組み込みの重要な違いは C++の `main()` 関数の場所である（それぞれ Python のインタープリタと他のプログラムである）。Python を他のプログラムへ組み込む場合であっても、拡張モジュールは C/C++から Python のコードへアクセス可能にする最も優れた方法であり、拡張モジュールの使用が両方のモデルの核心であることに注意していただきたい。

わずかな例外を除いて、単一のエントリポイントを持ち動的に読み込まれるライブラリとしてビルドする。つまり変更時に他の拡張モジュールや `main()` を持つ実行可能ファイルを再ビルドする必要がない。

3 インストールなしのクイックスタート

Boost.Python を使い始めるのに「Boost をインストール」する必要はない。この説明では、必要なバイナリをすぐにビルドする Boost.Build プロジェクトを利用する。最初のテストは Boost.Python のビルドより少し長くかかるかもしれないが、この方法であれば特定のコンパイラ設定に対してどのライブラリバイナリを使用すべきかといった厄介ごとに悩むことなく、正しいコンパイラオプションをあなた自身が理解できることだろう。

注意

他のビルドシステムを使用して Boost.Python やその拡張をビルドすることは当然可能であるが、Boost では公式にはサポートしない。**「Boost.Python がビルドできないよ」問題の 99%以上は**、以下の説明を無視して**他のビルドシステムを使用したことが原因である**。

それでも他のビルドシステムを使用したい場合は、以下の説明に従って `bjam` にオプション `-a -ofilename` を付けて起動し実行するビルドコマンドをファイルにダンプすれば、あなたのビルドシステムで必要なことが分かる。

4 Boost.Python の以前のバージョンと Python 2.2 の組み合わせでテストを行っており互換性を損なうようなことはしていないと**考えて**いる。しかしながら Boost.Python の最新版では Python の 2.4 より前のバージョンに対してテストを行っていない可能性があり、Python 2.2 および 2.3 をサポートしているとは 100%言い切れない。

3.1 基本的な手順

1. Boost を入手する。Boost 導入ガイドの第 1 節、第 2 節（Unix/Linux、Windows）を見よ。
2. bjam ビルドドライバを入手する。Boost 導入ガイドの第 5 節（Unix/Linux、Windows）を見よ。
3. Boost をインストールした/libs/python/example/quickstart/ディレクトリに cd で移動する。小さなプロジェクト例がある。
4. bjam を起動する。すべてのテストターゲットをビルドするため、導入ガイド第 5 節の起動例にある「stage」引数を「test」に置き換える。またテストが生成した出力を見るため、引数に「--verbose-test」を追加する。

Windows の場合、bjam の起動は以下のようになる。

```
C:\boost_1_34_0\…\quickstart> bjam toolset=msvc --verbose-test test
```

Unix 系の場合はおそらく、

```
~/boost_1_34_0/…/quickstart$ bjam toolset=gcc --verbose-test test
```

Windows ユーザへの注意

簡単のために、このガイドの残りの部分ではパス名によりなじみのあるバックスラッシュではなく、Unix スタイルのスラッシュを使用する。スラッシュはコマンドプロンプトウィンドウ以外のあらゆる場所で機能するはずである（コマンドプロンプトだけはバックスラッシュを使用しなければならない）。

ここまでの手順がうまくいったら、extending という名前の拡張モジュールのビルドが終わり、test_extending.py という Python スクリプトが走ってテストも完了しているはずである。また、Python を組み込む embedding という簡単なアプリケーションもビルド、起動する。

3.2 問題が起きた場合

コンパイラやリンカのエラーメッセージが大量に表示された場合、Boost.Build が Python のインストール情報を見つけられていない可能性が高い。bjam を起動する最初の数回、--debug-configuration オプションを bjam に渡して Boost.Build が Python のインストール構成部分をすべて正しく見つけられているか確認することだ。失敗している場合は、以下の Boost.Build を設定するの節を試すとよい。

それでもなお問題が解決しない場合は、以下のメーリングリストに手助けしてくれる人がいるかもしれない。

- Boost.Build に関する話題は Boost.Build のメーリングリスト
- Boost.Python に固有の話題は Python の C++ Sig

3.3 すべて問題ない場合

おめでとう！ Boost.Python に慣れていなければ、この時点でしばらくビルドに関することを忘れ、チュートリアルやリファレンスを通じてライブラリの学習に集中するとよいかもしれない。quickstart プロジェクトに変更を加えて、API について学んだことを十分に試してみるのもよい。

3.4 サンプルプロジェクトを変更する

拡張モジュールを（Boost ディストリビューション内の 1 つのソースファイルである）extending.cpp 内に限定し、これを extending としてインポートすることに満足しているのであれば、ここでやめてしまってもよい。しかしながら少しぐらいは変更したいと思うことだろう。Boost.Build について詳しく学ぶことなく先に進む方法はある。

今ビルドしたプロジェクトは現在のディレクトリにある 2 つのファイルで規定されている。boost-build.jam は Boost ビルドシステムのコードの場所を bjam に指定する。Jamroot はビルドしたターゲットを記述する。これらのファイルにはコメントを大量に書いてあるので、変更は容易なはずである。ただし空白の保持には注意していただきたい。;のような区切り文字は前後に空白がなければ bjam は認識しない。

プロジェクトの場所を変更する

Boost ディストリビューションに変更が生じないよう、このプロジェクトをどこか他の場所にコピーしたいと思うことだろう。単純に次のようにする。

1. libs/python/example/quickstart/ディレクトリ全体を新しいディレクトリにコピーする。
2. boost-build.jam および Jamroot の新コピーにおいて、ファイルの先頭付近で相対パスを探し（コメントで分かりやすくマークしてある）、ファイルが libs/python/example/quickstart ディレクトリ内の元の場所にあったときと同様に Boost ディストリビューションを指すように編集する。

例えばプロジェクトを/home/dave/boost_1_34_0/libs/python/example/quickstart から/home/dave/my-project へ移動したとすると、boost-build.jam の最初のパスは

```
../../../../tools/build/v2
```

次のように変更する。

```
/home/dave/boost_1_34_0/tools/build/v2
```

また Jamroot の最初のパスは

```
../../../..
```

次のように変更する。

```
/home/dave/boost_1_34_0
```

新しいソースファイルを追加するか既存ファイルの名前を変更する

拡張モジュールや組み込みアプリケーションのビルドに必要なファイルの名前は、Jamroot 内にそれぞれ extending.cpp、embedding.cpp の右に並べて書く。各ファイル名の前後に空白を入れるのを忘れてはならない。

```
… file1.cpp file2.cpp file3.cpp …
```

当然ながら、ソースファイルの名前を変更したければ Jamroot 内の名前を編集して Boost.Build に通知する。

拡張モジュールの名前を変更する

拡張モジュールの名前は以下の 2 つで決まる。

1. Jamroot 内の python-extension 直後の名前
2. extending.cpp 内で BOOST_PYTHON_MODULE に渡した名前

拡張モジュールの名前を extending から hello に変更するには、Jamroot を編集して次の行を

```
python-extension extending : extending.cpp ;
```

以下のように変更する。

```
python-extension hello : extending.cpp ;
```

また extending.cpp を編集して次の行を

```
BOOST_PYTHON_MODULE(extending)
```

以下のように変更する。

```
BOOST_PYTHON_MODULE(hello)
```

4 システムに Boost.Python をインストールする

Boost.Python は（ヘッダオンリーライブラリとは逆に）個別にコンパイルが必要なライブラリであるため、ユーザは Boost.Python ライブラリバイナリのサービスに依存する。

Boost.Python ライブラリのバイナリを普通にインストールする必要がある場合、Boost の導入ガイドを見ればその作成手順が一通り分かるだろう。ソースからバイナリをビルドする場合、Boost の全バイナリではなく Boost.Python のバイナリだけがビルドされるよう、

bjam に--with-python 引数を（あるいは configure に--with-libraries=python 引数を）渡すとよい。

5 Boost.Build を設定する

Boost.Build のリファレンスマニュアルにあるとおり、ビルドシステムで利用可能なツールとライブラリの指定はホームディレクトリ[5]の user-config.jam で行う。user-config.jam を作成・編集して Python の起動、ヘッダのインクルード、ライブラリのリンクについての方法を Boost.Build に指定する必要があるかもしれない。

Unix 系 OS のユーザ

Unix 系 OS を使用しており Boost の configure スクリプトを走らせた場合、user-config.jam が生成されている可能性がある。[6] configure/make シーケンスが成功して Boost.Python のバイナリがビルドされていれば、user-config.jam はおそらく既に正しい状態になっている。

Python を「標準的な」形でインストールしたのであれば、特に行うことはない。user-config.jam で python を設定していない（かつ Boost.Build コマンドラインで--without-python を指定していない）のであれば、Boost.Build は自動的に以下と等価なことを行い、最も適切な場所から Python を自動的に探し出す。

```
import toolset : using ;
using python ;
```

ただしこれが行われるのは Boost.Python のプロジェクトファイルを使用した場合だけである（例えば quickstart の方法のように別のプロジェクトから参照される場合）。個別にコンパイルした Boost.Python バイナリにリンクする場合は、上に挙げた最小限のおまじないで user-config.jam をセットアップしなければならない。

5.1 Python の設定引数

Python を複数バージョンインストールしている場合や Python を通常でない方法でインストールした場合は、以下の省略可能な引数のいずれか、またはすべてを using python に与えなければならない可能性がある。

version	使用する Python のバージョン。（メジャー）.（マイナー）の形式でなければらない（例：2.3）。サブマイナーバージョンは含めてはならない（2.5.1 は**不可**）。複数のバージョンの Python をインストールした場合、大抵は version が省略不可能な唯一の引数となる。
cmd-or-prefix	Python インタープリタを起動するコマンド。または Python のライブラリやヘッダファイルのインストール接頭辞。このパラメータの使用は、適切な Python 実行可能ファイルがない場合に限定すること。
includes	Python ヘッダの#include パス。通常、version と cmd-or-prefix から適切なパスが推測される。
libraries	Python ライブラリのバイナリへのパス。MacOS/Darwin では Python フレームワークのパスを渡してもよい。通常、version と cmd-or-prefix から適切なパスが推測される。
condition	指定する場合は、Boost.Build が使用する Python の設定を選択するときのビルド設定にマッチした Boost.Build

5 Windows でホームディレクトリを確認するには、コマンドプロンプトウィンドウで ECHO %HOMEDRIVE%%HOMEPATH%とタイプする。

6 configure はホームディレクトリにある既存の user-config.jam について、（あれば）バックアップを作成した後で上書きする。

	パラメータの集合でなければならない。詳細は以下の例を見よ。
extension-suffix	拡張モジュール名の真のファイル拡張子の前に追加する文字列。ほとんどの場合、この引数を使用する必要はない。大抵の場合、この接尾辞を使用するのは Windows において Python のデバッグビルドをターゲットにする場合だけであり、<python-debugging>機能の値に基づいて自動的に設定される。しかしながら少なくとも 1 つの Linux のディストリビューション（Ubuntu Feisty Fawn）では python-dbg は特殊な設定がなされており、この種の接頭辞を使用しなければならない。

5.2 例

以下の例では大文字小文字の区別や、**特に空白**が重要である。

Python 2.5 と Python 2.4 の両方をインストールしている場合、user-config.jam を次のようにしておく。

```
using python : 2.5 ;  # 両方のバージョンの Python を有効にする

using python : 2.4 ;  # python 2.4 でビルドする場合は、python=2.4 を
                      # コマンドラインに追加する。
```

最初のバージョン設定（2.5）が既定となる。Python 2.4 についてビルドする場合は bjam コマンドラインに python=2.4 を追加する。

Python を通常でない場所にインストールしている場合、cmd-or-prefix 引数にインタープリタへのパスを与えるとよい。

```
using python : : /usr/local/python-2.6-beta/bin/python ;
```

特定のツールセットに対して個別の Python ビルドを置いている場合、condition 引数にそのツールセットを与えるとよい。

```
using python ;  # 通常のツールセットで使用

# Intel C++ツールセットで使用
using python
     : # version
     : c:\\Devel\\Python-2.5-IntelBuild\\PCBuild\\python # cmd-or-prefix
     : # includes
     : # libraries
     : <toolset>intel # condition
     ;
```

Python のソースをダウンロードし、Windows 上でソースから通常版と「Python デバッグ」ビルド版の両方をビルドした場合、次のようにするとよい。

```
using python : 2.5 : C:\\src\\Python-2.5\\PCBuild\\python ;
using python : 2.5 : C:\\src\\Python-2.5\\PCBuild\\python_d
  : # includes
  : # libs
  : <python-debugging>on ;
```

Windows でビルドした bjam では、Windows と Cygwin の両方の Python 拡張をビルド・テストできるよう user-config.jam をセッ

トアップできる。Cygwin の Python インストールに対して condition 引数に<target-os>cygwin を渡すだけでよい。

```
# Windows 版
using python ;

# Cygwin 版
using python : : c:\\cygwin\\bin\\python2.5 : : : <target-os>cygwin ;
```

ビルドリクエストに target-os=cygwin と書くと、Cygwin 版の Python でビルドが行われる。[7]

```
bjam target-os=cygwin toolset=gcc
```

他の方法でも同様に動作すると思う（Windows 版の Python をターゲットにして Cygwin 版の bjam を使用する場合）が、本稿執筆時点ではそのような組み合わせのビルドに対するツールセットのサポートにはバグがあるようだ。

Boost.Build がターゲットを選択する方法が原因で、ビルドリクエストは完全に明確にしなければならないということに注意していただきたい。例えば、次のようであるとすると、

```
using python : 2.5 ; # 通常の Windows ビルドの場合
using python : 2.4 : : : : <target-os>cygwin ;
```

以下の方法でビルドするとエラーになる。

```
bjam target-os=cygwin
```

代わりに、こう書く必要がある。

```
bjam target-os=cygwin/python=2.4
```

6 Boost.Python ライブラリのバイナリを選択する

（Boost.Build に自動的に正しいライブラリを構築、リンクさせる代わりに）ビルド済みの Boost.Python ライブラリを使用する場合、どれをリンクするか考える必要がある。Boost.Python バイナリには動的版と静的版がある。アプリケーションに応じて注意して選択しなければならない。[8]

6.1 動的バイナリ

動的ライブラリは最も安全で最も汎用性の高い選択である。

- 与えられたツールセットでビルドしたすべての拡張モジュールが、ライブラリコードの単一のコピーを使用する。[9]

7 <target-os>cygwin 機能は gcc ツール群の<flavor>cygwin サブ機能とは別物であることに注意していただきたい。MinGW GCC もインストールしている場合は、両者を明示的に扱わなければならない。

8 Boost.Python の静的ビルドと動的ビルドを区別する方法：Windows の場合／Unix 系の場合

9 ほとんどの Unix/Linux 系プラットフォームでは動的に読み込んだオブジェクトを共有するため、異なるコンパイラツールセットでビルドした拡張

- ライブラリには型変換レジストリが含まれる。すべての拡張モジュールが単一のレジストリを共有するので、ある動的に読み込んだ拡張モジュールで Python へエクスポートしたクラスのインスタンスを、別のモジュールでエクスポートした関数へ渡すことができる。

6.2 静的バイナリ

以下のいずれかの場合は Boost.Python の静的ライブラリを使用するのが適切である。

- Python を拡張していて、動的に読み込んだ拡張モジュールで他の Boost.Python 拡張モジュールから使用する必要のない型をエクスポートしており、かつそれらの間でコアライブラリコードが複製されても問題ない場合。
- Python をアプリケーションに組み込んでおり、かつ以下のいずれかの場合。
  - MacOS か AIX 以外の Unix 系 OS をターゲットにしていて、動的に読み込んだ拡張モジュールから実行可能ファイルの一部である Boost.Python ライブラリシンボルが「見える」場合。
  - またはアプリケーションに何らかの Boost.Python 拡張モジュールを静的にリンクしており、かつ動的に読み込んだ Boost.Python 拡張モジュールが静的にリンクした拡張モジュールでエクスポートした型を使用 可能でも問題ない場合（あるいはその逆）。

7 #include に関すること

1. Boost.Python を使用するプログラムの翻訳単位で直接 `#include "python.h"` と書きたくなったら、代わりに `#include "boost/python/detail/wrap_python.hpp"` を使用せよ。こうすることで Boost.Python を使用するのに必要ないくつかの事柄が適当に処理される（その中の 1 つを次節で見る）。
2. `wrap_python.hpp` の前にシステムヘッダをインクルードしないよう注意せよ。この制限は実際には Python によるものであり、より正確には Python とオペレーティングシステムの相互作用によるものである。詳細は http://docs.python.org/ext/simpleExample.html[10]を見よ。

8 Python のデバッグビルド

Python は特殊な「python debugging」設定でビルドすることで、拡張モジュールの開発者にとって非常に有用なチェックとインストゥルメント（instrumentation）を追加できる。デバッグ設定が使用するデータ構造は追加のメンバの含んでいるため、**python debugging を有効にした Python 実行可能ビルドを、有効にせずにコンパイルしたライブラリや拡張モジュールとともに使用することはできない（その逆も同様である）。**

モジュールを同一の Python インスタンスに読み込んだとき常に異なる Boost.Python ライブラリのコピーを使用するか定かではない。それらのコンパイラが互換性のある ABI を有しているのであれば、2 つのライブラリでビルドした拡張モジュールは相互運用可能なので、別のライブラリを使用しないほうが望ましい。そうでなければ拡張モジュールと Boost.Python はクラスレイアウト等が異なるため、大惨事となるかもしれない。何が起こるか確認する実験を行ってくれる人がいれば幸いである。

10 訳注　日本語訳は http://docs.python.jp/2/extending/extending.html#extending-simpleexample（Python 2.x の場合）。

Python のビルド済み実行可能の「python debugging」版はほとんどの Python ディストリビューションでは提供されておらず[11]、それらのビルドをユーザに強制したくないので、debug ビルド（既定）において python debugging を自動的に有効化することはない。代わりに python-debugging という特殊なビルドプロパティを用意しており、これを使用すると適切なプリプロセッサシンボルが定義され正しいライブラリがリンク先として選択される。

Unix 系プラットフォームでは、デバッグ版 Python のデータ構造は Py_DEBUG シンボルを定義したときのみ使用される。多くの Windows コンパイラでは、プリプロセッサシンボル _DEBUG を付けてビルドすると、Python は既定では Python DLL の特殊デバッグ版へのリンクを強制する。このシンボルは Python の有無とは別にごくありふれたものであるので、Boost.Python は boost/python/detail/wrap_python.hpp で Python.h をインクルードするときから BOOST_DEBUG_PYTHON が定義されるまでの間、一時的に _DEBUG を未定義にする。結論としては「python debugging」が必要で Boost.Build を使用しない場合は、BOOST_DEBUG_PYTHON が定義されているのを確認することである。そうでなければ python debugging は無効になる。

9 Boost.Python をテストする

Boost.Build の完全なテストスイートを走らせるには、Boost ディストリビューションの libs/python/test サブディレクトリで bjam を起動する。

10 MinGW（および Cygwin の-mno-cygwin）の GCC ユーザに対する注意

Python の 2.4.1 より前のバージョンを MinGW の 3.0.0（binutils-2.13.90-20030111-1）より前のバージョンで使用する場合は、MinGW 互換バージョンの Python ライブラリを作成する必要がある（Python に付属のものは Microsoft 互換のリンカでしか動作しない）。『Python モジュールのインストール』の「拡張モジュールのビルド: 小技と豆知識」—「Windows で非 Microsoft コンパイラを使ってビルドするには」の節に従い、libpythonXX.a を作成する。XX はインストールした Python のメジャーバージョンとマイナーバージョン番号である。

11 Unix 系列のプラットフォームでは、debugging python と関連ライブラリは Python のビルド設定で--with-pydebug を追加するとビルドされる。Windows では Python のデバッグ版は、Python の完全なソースコードディストリビューションの PCBuild サブディレクトリにある Visual Studio プロジェクトの「Win32 Debug」ターゲットから生成される。

リファレンスマニュアル

コンセプト

CallPolicies コンセプト

はじめに

CallPolicies コンセプトのモデルは、Boost.Python が生成した Python の呼び出し可能オブジェクトの振る舞いを、関数やメンバ関数ポインタのようなラップした C++オブジェクトに対して特殊化するのに使用する。以下の 4 つの振る舞いを提供する。

1. `precall` — ラップしたオブジェクトが呼び出される前の Python 引数タプルの管理
2. `result_converter` — C++戻り値の処理
3. `postcall` — ラップしたオブジェクトが呼び出された後の Python 引数タプルと戻り値の管理
4. `extract_return_type` — 与えられたシグニチャ型並びから戻り値の型を抽出するメタ関数

CallPolicies の合成

ユーザが同じ呼び出し可能オブジェクトで複数の CallPolicies モデルを使用可能にするため、Boost.Python の CallPolicies クラステンプレートは再帰的に合成可能な数珠繋ぎのインターフェイスを提供する。このインターフェイスは省略可能なテンプレート引数 `Base` を取り、既定は `default_call_policies` である。慣習に従って、`Base` の `precall` 関数はより外側のテンプレートが与えた `precall` 関数の**後で**呼び出され、`Base` の `postcall` 関数はより外側のテンプレートの `postcall` 関数より**前に**呼び出される。より外側のテンプレートで `result_converter` が与えられた場合、`Base` で与えた `result_converter` は**上書きされる**。例えば `return_internal_reference` を見よ。

コンセプトの要件

CallPolicies コンセプト

以下の表で `x` は CallPolicies のモデルである型 `P` のオブジェクト、`a` は Python の引数タプルオブジェクトを指す `PyObject*`、`r` は「予備的な」戻り値オブジェクトを参照する `PyObject*`を表す。

式	型	結果・セマンティクス
`x.precall(a)`	`bool` へ変換可能	失敗時は `false` および `PyErr_Occurred() != 0`、それ以外は `true`。
`P::result_converter`	ResultConverterGenerator のモデル	「予備的な」戻り値オブジェクトを生成する MPL の単項メタ関

式	型	結果・セマンティクス
		数クラス。
`x.postcall(a, r)`	`PyObject*`へ変換可能	失敗時は `0` および `PyErr_Occurred() != 0`。例外送出中であっても「参照を保持」しなければならない。言い換えると、`r` を返さない場合はその参照カウントを減らさなければならず、別の生存しているオブジェクトを返す場合はその参照カウントを増やさなければならない。
`P::extract_return_type`	Metafunction のモデル	与えられたシグニチャから戻り値の型を抽出する MPL の単項 Metafunction。既定では `mpl::front` から導出する。

CallPolicies のモデルは CopyConstructible であることが要求される。

Dereferenceable コンセプト

はじめに

Dereferenceable 型のインスタンスは lvalue へアクセスするポインタのように使用できる。

コンセプトの要件

Dereferenceable コンセプト

以下の表で `T` は Dereferenceable のモデル、`x` は型 `T` のオブジェクトを表す。またポインタはすべて Dereferenceable とする。

式	結果	操作上のセマンティクス
`get_pointer(x)`	`pointee<T>::type*`へ変換可能	`&*x` かヌルポインタ

Extractor コンセプト

はじめに

Extractor は、Boost.Python が Python オブジェクトから C++オブジェクトを抽出するのに使用するクラスである。典型的には、「伝統的な」Python 拡張型について `from_python` 変換を定義するのに使用する。

コンセプトの要件

Extractor コンセプト

以下の表で `X` は Extractor のモデル、`a` は Python オブジェクト型のインスタンスを表す。

式	型	セマンティクス
`X::execute(a)`	非 void	抽出する C++オブジェクトを返す。`execute` 関数は多重定義してはならない。
`&a.ob_type`	`PyTypeObject**`	`PyObject` とレイアウト互換なオブジェクトの `ob_type` フィールドを指す。

注意事項

簡単に言うと、Extractor の `execute` メンバは多重定義のない、Python オブジェクト型を 1 つだけ引数にとる静的関数でなければならない。`PyObject` から公開派生した（かつあいまいでない継承関係にある）型、および `PyObject` とレイアウト互換な POD 型もこの Python オブジェクト型に含まれる。

HolderGenerator コンセプト

はじめに

HolderGenerator はその引数のインスタンスを、ラップした C++クラスインスタンス内に保持するのに適した型を返す単項メタ関数クラスである。

コンセンプトの要件

HolderGenerator コンセプト

以下の表で `G` は HolderGenerator のモデル型、`X` はクラス型を表す。

式	要件
`G::apply<X>::type`	型 `X` のオブジェクトを保持できる `instance_holder` の具象派生クラス。

ResultConverter/ResultConverterGenerator コンセプト

はじめに

型 `T` に対する ResultConverter は、型 `T` の C++戻り値を `to_python` 変換するのにそのインスタンスが使用可能な型である。ResultConverterGenerator は、C++関数の戻り値型が与えられたときにその型に対する ResultConverter を返す MPL の単項メタ関数クラスである。Boost.Python における ResultConverter は一般的にライブラリの変換器レジストリを探索して適切な変換器を探すが、レジストリを使用しない変換器もありうる。

コンセプトの要件

ResultConverter コンセプト

以下の表で `C` は型 `R` に対する ResultConverter、`c` は型 `C` のオブジェクト、`r` は型 `R` のオブジェクトを表す。

式	型	セマンティクス
`C c;`		`C` のオブジェクトを構築する。
`c.convertible()`	`bool` へ変換可能	`R` の値から Python オブジェクトへの変換が不可能な場合は `false`。
`c(r)`	`PyObject*`へ変換可能	`r` に対応する Python オブジェクトへのポインタ。`r` が `to_python` 変換不可能な場合は `0` で、`PyErr_Occurred`は非 0 を返すはずである。
`c.get_pytype()`	`PyTypeObject const*`	変換の結果に対応する Python の型オブジェクトへのポインタか `0`。ドキュメンテーションの生成に使用する。`0` を返した場合はドキュメンテーション内で生成された型は **object** になる。

ResultConverterGenerator コンセプト

以下の表で `G` は ResultConverterGenerator 型、`R` は C++関数の戻り値型。

式	要件
`G::apply<R>::type`	`R` に対する ResultConverter 型。

ObjectWrapper/TypeWrapper コンセプト

はじめに

Python オブジェクトを管理するクラスと Python 風の構文をサポートするクラスを表現する、2 つのコンセプトを定義する。

コンセプトの要件

ObjectWrapper コンセプト

ObjectWrapper コンセプトのモデルは公開アクセス可能な基底クラスとして `object` を持ち、特殊な構築の振る舞いとメンバ関数（テンプレートメンバ関数であることが多い）による便利な機能を提供する。戻り値の型 `R` 自身が TypeWrapper である場合を除いて、次のメンバ関数呼び出し形式は

```
x.some_function(a1, a2,...an)
```

常に以下と同じセマンティクスを持つ。

```
extract<R>(x.attr("some_function")(object(a1), object(a2),...object(an)))()
```

`R` が TypeWrapper である場合、戻り値の型は直接以下により構築する。

```
x.attr("some_function")(object(a1), object(a2),...object(an)).ptr()
```

（ただし以下の注意も見よ。）

TypeWrapper コンセプト

TypeWrapper は個々の Python の型 X と一体となった ObjectWrapper の改良版である。TypeWrapper として T があるとすると、

```
T(a1, a2,...an)
```

上記の合法なコンストラクタ式は、以下に相当する引数列で X を呼び出した結果を管理する新しい T オブジェクトを構築する。

```
object(a1), object(a2),...object(an)
```

ラップした C++関数の引数、あるいは extract<>のテンプレート引数として使用した場合、対応する Python 型のインスタンスだけがマッチするとみなされる。

注意

戻り値の型が TypeWrapper である場合の特殊なメンバ関数の呼び出し規則は結論として、返されたオブジェクトが不適切な型の Python オブジェクトを管理している可能がある。これは大抵の場合、深刻な問題とはならない（最悪の場合の結果は、エラーが実行時に検出される場合、つまり他のあらゆる場合よりも少し遅いタイミングだ）。このようなことが起こる例として、dict のメンバ関数 items が list 型のオブジェクトを返すことに注意していただきたい。今、ユーザがこの dict の派生クラスを Python 内で定義すると、

```
>>> class mydict(dict):
...     def items(self):
...         return tuple(dict.items(self)) # タプルを返す
```

mydict のインスタンスは dict のインスタンスでもあるため、ラップした C++関数の引数として使用すると、boost::python::dict は Python の mydict 型オブジェクトも受け取る。このオブジェクトに対して items()を呼び出すと、実際には Python のタプルを保持した boost::python::list のインスタンスが返る。このオブジェクトで続けて list のメソッド（例えば append 等、変更を伴う操作）を使用すると、同じことを Python で行った場合と同様の例外が送出する。

高水準コンポーネント

ヘッダ<boost/python/class.hpp>および<boost/python/class_fwd.hpp>

はじめに

<boost/python/class.hpp>は、ユーザが C++クラスを Python へエクスポートするためのインターフェイスを定義する。class_クラステンプレートを宣言し、その引数はエクスポートするクラス型である。また init、optional および bases ユーティリティクラステンプレートもエクスポートし、これらは class_クラステンプレートと組み合わせて使用する。

<boost/python/class_fwd.hpp>には class_クラステンプレートの先行宣言がある。

クラス

class_<T, Bases, HeldType, NonCopyable>クラステンプレート

第 1 引数として渡した C++型に対応する Python クラスを作成する。引数は 4 つあるが、必須なのは 1 番目だけである。3 つの省略可能な引数は**どのような順序でもよく**、Boost.Python は引数の型から役割を判断する。

テンプレート引数	要件	セマンティクス	既定
T	クラス型。	ラップするクラス。	
Bases	それ以前にエクスポートした T の C++基底クラス群を指定する bases<...>の特殊化。[12]	ラップした T インスタンスからその直接的または間接的な基本型それぞれへの from_python 変換を登録する。多態的な各基底型 B について、間接的に保持されたラップした B インスタンスから T への変換を登録する。	bases<>
HeldType	T、T の派生クラス、または pointee<HeldType>::type が T か T の派生クラスである Dereferenceable 型。	T のコンストラクタを呼び出したとき、または T や T* を ptr、ref、あるいは return_internal_reference 等の CallPolicies を使用せずに Python へ変換したとき、T のインスタンスをラップする Python オブジェクトへ実際に組み込む型を指定する。詳細は後述する。	T
NonCopyable	与えられた場合、boost::noncopyable でなければならない。	T のインスタンスをコピーする to_python 変換の自動的な登録を抑止する。T が公開アクセス可能なコピーコンストラクタを持たない場合に必要である。	boost::noncopyable 以外の未規定の型。

HeldType のセマンティクス

1. HeldType が T から派生している場合、そのエクスポートしたコンストラクタは、HeldType インスタンスを持つ Python オブジェクトを逆向きに参照する PyObject*を第 1 引数に受け取らなければならない（例）。この引数は def(init_expr)に渡される *init-expression* には含まれず、T の Python インスタンスが作成されるときユーザが明示的に渡すこともない。このイディオムにより、Python 側でオーバーライドする C++仮想関数が Python オブジェクトにアクセス可能となり、Python のメソッドが呼び出し可能となる。Boost.Python は HeldType 引数を要求するラップした C++関数に T のラップしたインスタンスを渡すための変換器を自動的に登録する。
2. Boost.Python は T のラップしたインスタンスが HeldType 型の引数として渡すことを認めているため、HeldType のスマートポインタを指定することでユーザは T へのスマートポインタを受け取るところに Python の T インスタンスを渡すことができる。std::auto_ptr や boost::shared_ptr<>といった対象の型を指す入れ子の element_type 型を持つスマートポインタは自動的にサポートされる。[13]pointee<HeldType>を特殊化することで、他のスマートポインタ型もサポートされる。
3. 上の 1 で述べたとおり、HeldType が T の派生型に対するスマートポインタの場合、HeldType のエクスポートしたコンストラ

12 「それ以前にエクスポートした」とは bases 内の各 B について、class_<B, ...>が構築済みでなければならないという意味である。
```
class_<Base>("Base");
class_<Derived, bases<Base> >("Derived");
```

13 訳注 std::shared_ptr（C++11 以降）も自動的にサポートされます。

クタすべてで PyObject*を第 1 引数として与えなければならない。

4. 上記 1 および 3 以外でユーザは has_back_reference<T>を特殊化することで、T 自身が第 1 引数である PyObject*で初期化されることをオプション的に指定できる。

class_クラステンプレートの概要

```
namespace boost { namespace python
{
  template <class T
      , class Bases = bases<>
            , class HeldType = T
            , class NonCopyable = unspecified
          >
  class class_ : public object
  {
    // 既定の__init__を使用するコンストラクタ
    class_(char const* name);
    class_(char const* name, char const* docstring);

    // 既定でない__init__を指定するコンストラクタ
    template <class Init>
    class_(char const* name, Init);
    template <class Init>
    class_(char const* name, char const* docstring, Init);

    // 追加の__init__関数のエクスポート
    template <class Init>
    class_& def(Init);

    // メソッドの定義
    template <class F>
    class_& def(char const* name, F f);
    template <class Fn, class A1>
    class_& def(char const* name, Fn fn, A1 const&);
    template <class Fn, class A1, class A2>
    class_& def(char const* name, Fn fn, A1 const&, A2 const&);
    template <class Fn, class A1, class A2, class A3>
    class_& def(char const* name, Fn fn, A1 const&, A2 const&, A3 const&);

    // メソッドをstaticとして宣言
    class_& staticmethod(char const* name);

    // 演算子のエクスポート
    template <unspecified>
    class_& def(detail::operator_<unspecified>);

    // 生の属性の変更
    template <class U>
    class_& setattr(char const* name, U const&);

    // データメンバのエクスポート
    template <class D>
    class_& def_readonly(char const* name, D T::*pm);
```

```
    template <class D>
    class_& def_readwrite(char const* name, D T::*pm);

    // staticデータメンバのエクスポート
    template <class D>
    class_& def_readonly(char const* name, D const& d);
    template <class D>
    class_& def_readwrite(char const* name, D& d);

    // プロパティの作成
    template <class Get>
    void add_property(char const* name, Get const& fget, char const* doc=0);
    template <class Get, class Set>
    void add_property(
        char const* name, Get const& fget, Set const& fset, char const* doc=0);

    template <class Get>
    void add_static_property(char const* name, Get const& fget);
    template <class Get, class Set>
    void add_static_property(char const* name, Get const& fget, Set const& fset);

    // pickleのサポート
    template <typename PickleSuite>
    self& def_pickle(PickleSuite const&);
    self& enable_pickling();
  };
}}
```

class_クラステンプレートのコンストラクタ

```
class_(char const* name);
class_(char const* name, char const* docstring);
template <class Init>
class_(char const* name, Init init_spec);
template <class Init>
class_(char const* name, char const* docstring, Init init_spec);
```

要件

name は Python の識別子の名前付け規約にしたがった ntbs。*docstring* が与えられた場合は ntbs でなければならない。*init_spec* が与えられた場合、特殊な列挙定数 `no_init` か `T` と互換性のある *init-expression* のいずれかでなければならない。

効果

名前 *name* の Boost.Python 拡張クラスを保持する `class_` オブジェクトを構築する。現在のスコープにおいて属性 *name* を新しい拡張クラスに束縛する。

- *docstring* が与えられた場合、その値を拡張クラスの `__doc__` 属性に束縛する。
- *init_spec* が `no_init` である場合、常に Python 例外を投げる特殊な `__init__` 関数を生成する。それ以外の場合は `this->def(init_spec)` を呼び出す。
- *init_spec* が与えられなかった場合、`this->def(init<>())` を呼び出す。

根拠

必要な T インスタンスを作成する__init__関数をエクスポートせずにラップしたメンバ関数を呼び出すことによって発生する、よくある実行時エラーを避けられるよう、class_<>コンストラクタ内でコンストラクタ引数を指定できる。

クラステンプレート class_の変更関数

```
template <class Init>
class_& def(Init init_expr);
```

要件

init_expr は T と互換性のある *init-expression* の結果。

効果

Init の合法な接頭辞 *P* それぞれについて、*P* を引数として受け取る拡張クラスに__init__(...)関数の多重定義を追加する。生成された各多重定義は、上述のセマンティクスに従って *init_expr* の呼び出しポリシーのコピーを使用して HeldType のオブジェクトを構築する。*Init* の合法な接頭辞の最長のものが *N* 個の型を有しており *init_expr* が *M* 個のキーワードを保持しているとすると、各多重定義の先頭の *N* – *M* 個の引数に使用される。

戻り値

*this

根拠

ユーザはクラスのコンストラクタを容易に Python へエクスポートできる。

```
template <class F>
class_& def(char const* name, Fn fn);
template <class Fn, class A1>
class_& def(char const* name, Fn fn, A1 const& a1);
template <class Fn, class A1, class A2>
class_& def(char const* name, Fn fn, A1 const& a1, A2 const& a2);
template <class Fn, class A1, class A2, class A3>
class_& def(char const* name, Fn fn, A1 const& a1, A2 const& a2, A3 const& a3);
```

要件

name は Python の識別子の名前付け規約にしたがった ntbs。

- *a1* が *overload-dispatch-expression* の結果である場合、有効なのは 2 番目の形式のみであり *fn* は引数長が *A1* の最大引数長と同じである関数かメンバ関数へのポインタでなければならない。

効果

Fn の引数型列の接頭辞 *P* それぞれについて、その長さが *A1* の最小引数長であるものから、拡張クラスに *name*(...)メソッドの多重定義を追加する。生成された各多重定義は、*a1* の呼び出しポリシーのコピーを使用して *a1* の *call-expression* を *P* とともに呼び出す。*A1* の合法な接頭辞の最長のものが *N* 個の型を有しており *a1* が *M* 個のキーワードを保持しているとすると、各多重定義の先頭の *N* – *M* 個の引数に使用される。

- それ以外の場合、*fn* に対してメソッドの多重定義を 1 つ構築する。*Fn* は null であってはならない。

- *fn* が関数ポインタである場合、第 1 引数は `U`、`U cv&`、`U cv*`、`U cv* const&`のいずれか（`T*`が `U*`に変換可能とする）でなければならない。*a1* から *a3* が与えられた場合、下表から任意の順番であってよい。
- 上記以外で *fn* がメンバ関数へのポインタである場合、参照先は `T` かその公開基底クラスでなければならない。*a1* から *a3* が与えられた場合、下表から任意の順番であってよい。
- それ以外の場合、*Fn* は `object` かその派生型でなければならない。*a1* から *a2* が与えられた場合、下表の 2 行目までから任意の順番であってよい。*fn* が呼び出し可能でなければならない。

名前	要件・型特性	効果
docstring	ntbs	値は結果の多重定義メソッドの`__doc__`属性に束縛される。それ以前の多重定義でドキュメンテーション文字列が与えられている場合は、改行 2 文字と新しいドキュメンテーション文字列がそこに追加される。
policies	CallPolicies のモデル	結果の多重定義メソッドの呼び出しポリシーとしてコピーが使用される。
keywords	*fn* の引数長を超えることがないことを指定する *keyword-expression* の結果。	結果の多重定義メソッドの呼び出しポリシーとしてコピーが使用される。

戻り値

`*this`

```
class_& staticmethod(char const* name);
```

要件

name は Python の識別子の名前付け規約にしたがった ntbs であり、多重定義がすべて定義済みのメソッドの名前。

効果

既存の名前 *x* の属性を Python の `staticmethod(x)` 呼び出し結果で置換する。当該メソッドが静的でありオブジェクトを渡さないことを指定する。これは以下の Python 文と等価である。

```
setattr(self, name, staticmethod(getattr(self, name)))
```

注意

`staticmethod(name)` 呼び出し後に `def(name,...)` を呼び出すと、`RuntimeError` を送出する。

戻り値

`*this`

```
template <unspecified>
class_& def(detail::operator_<unspecified>);
```

効果

ここに示す Python の特殊関数を追加する。

戻り値

*this

```
template <class U>
class_& setattr(char const* name, U const& u);
```

要件

name は Python の識別子の名前付け規約にしたがった ntbs。

効果

u を Python へ変換し、拡張クラスの属性辞書に追加する。

```
PyObject_SetAttrString(this->ptr(), name, object(u).ptr());
```

戻り値

*this

```
template <class Get>
void add_property(char const* name, Get const& fget, char const* doc=0);
template <class Get, class Set>
void add_property(
        char const* name, Get const& fget, Set const& fset, char const* doc=0);
```

要件

name は Python の識別子の名前付け規約にしたがった ntbs。

効果

新しい Python の property クラスインスタンスを作成し、object(fget)（2 番目の形式では object(fset)も）および（省略可能な）ドキュメンテーション文字列 *doc* をコンストラクタに渡す。最後にこのプロパティを構築中の Python のクラスオブジェクトに与えられた属性名 *name* で追加する。

戻り値

*this

根拠

ユーザは、Python の属性アクセス構文で呼び出せる関数を容易にエクスポートできる。

```
template <class Get>
void add_static_property(char const* name, Get const& fget);
template <class Get, class Set>
void add_static_property(char const* name, Get const& fget, Set const& fset);
```

要件

name は Python の識別子の名前付け規約にしたがった ntbs。

効果

Boost.Python.StaticProperty オブジェクトを作成し、object(fget)（2 番目の形式では object(fset)も）をコンストラクタに渡す。最後にこのプロパティを構築中の Python のクラスオブジェクトに与えられた属性名 *name* で追加する。StaticProperty は先頭の `self`

引数なしで呼び出せる Python の property クラスの特殊な派生クラスである。

戻り値

*this

根拠

ユーザは、Python の属性アクセス構文で呼び出せる関数を容易にエクスポートできる。

```
template <class D>
class_& def_readonly(char const* name, D T::*pm, char const* doc=0);
template <class D>
class_& def_readonly(char const* name, D const& d);
```

要件

name は Python の識別子の名前付け規約にしたがった ntbs。*doc* も ntbs。

効果

それぞれ、

```
this->add_property(name, make_getter(pm), doc);
```

および

```
this->add_static_property(name, make_getter(d));
```

戻り値

*this

根拠

ユーザは、Python から自然な構文で取得可能なクラスのデータメンバや自由変数を容易にエクスポートできる。

```
template <class D>
class_& def_readwrite(char const* name, D T::*pm, char const* doc=0);
template <class D>
class_& def_readwrite(char const* name, D& d);
```

効果

ぞれぞれ、

```
this->add_property(name, make_getter(pm), make_setter(pm), doc);
```

および

```
this->add_static_property(name, make_getter(d), make_setter(d));
```

戻り値

*this

根拠

ユーザは、Python から自然な構文で取得・設定可能なクラスのデータメンバや自由変数を容易にエクスポートできる。

```
template <typename PickleSuite>
class_& def_pickle(PickleSuite const&);
```

要件

PickleSuite は pickle_suite の公開派生型でなければならない。

効果

次の特殊な属性およびメソッドの合法な組み合わせを定義する：__getinitargs__、__getstate__、__setstate__、__getstate_manages_dict__、__safe_for_unpickling__、__reduce__

戻り値

*this

根拠

ユーザは、ラップしたクラスについて完全な pickle サポートを確立するための高水準インターフェイスを容易に使用できる。

```
class_& enable_pickling();
```

効果

__reduce__メソッドと__safe_for_unpickling__属性を定義する。

戻り値

*this

根拠

def_pickle()の軽量な代替。Python から pickle サポートの実装を有効にする。

bases<T1, T2, ...TN>クラステンプレート

class_<...>のインスタンス化において基底クラスのリストを記述するのに使用する MPL シーケンス。

bases クラステンプレートの概要

```
namespace boost { namespace python
{
  template <T1 = unspecified,...Tn = unspecified>
  struct bases
  {};
}}
```

例

次のような C++クラス宣言があるとすると、

```
class Foo : public Bar, public Baz
{
 public:
   Foo(int x, char const* y);
   Foo(double);

   std::string const& name() { return m_name; }
   void name(char const*);

   double value; // 公開データ
 private:
   ...
};
```

対応する Boost.Python 拡張クラスは以下のように作成できる。

```
using namespace boost::python;

class_<Foo,bases<Bar,Baz> >("Foo",
          "これは Foo のドキュメンテーション文字列。"
          "Foo 拡張クラスの記述がここに入る",

          init<int,char const*>(args("x","y"), "__init__ のドキュメンテーション文字列")
          )
   .def(init<double>())
   .def("get_name", &Foo::get_name, return_internal_reference<>())
   .def("set_name", &Foo::set_name)
   .def_readwrite("value", &Foo::value)
   ;
```

ヘッダ<boost/python/def.hpp>

はじめに

def() は現在のスコープで C++の関数や呼び出し可能オブジェクトを Python の関数としてエクスポートする関数である。

関数

def

```
template <class F>
void def(char const* name, F f);

template <class Fn, class A1>
void def(char const* name, Fn fn, A1 const&);

template <class Fn, class A1, class A2>
void def(char const* name, Fn fn, A1 const&, A2 const&);

template <class Fn, class A1, class A2, class A3>
```

```
void def(char const* name, Fn fn, A1 const&, A2 const&, A3 const&);
```

要件

name は Python の識別子の名前付け規約にしたがった ntbs。

- *Fn* が `object` かその派生型である場合、現在のスコープに多重定義の 1 つとして追加される。*fn* は呼び出し可能でなければならない。
- *a1* が *overload-dispatch-expression* の結果である場合、有効なのは 2 番目の形式のみであり *fn* は引数長が *A1* の最大引数長と同じ関数へのポインタかメンバ関数へのポインタでなければならない。

効果

Fn の引数型列の接頭辞 *P* それぞれについて、その長さが *A1* の**最小引数長**であるものから、現在のスコープに *name*(...) 関数の多重定義を追加する。生成された各多重定義は、*a1* の呼び出しポリシーのコピーを使用して *a1* の *call-expression* を *P* とともに呼び出す。*A1* の合法な接頭辞の最長のものが *N* 個の型を有しており *a1* が *M* 個のキーワードを保持しているとすると、各多重定義の先頭の *N* – *M* 個の引数に使用される。

それ以外の場合、*fn* は null でない関数へのポインタかメンバ関数へのポインタでなければならず、現在のスコープに *fn* の多重定義を 1 つ追加する。*a1* から *a3* のいずれかが与えられた場合、下表から任意の順番であってよい。

名前	要件・型の特性	効果
docstring	ntbs	値は結果の多重定義メソッドの`__doc__`属性に束縛される。
policies	CallPolicies のモデル	結果の多重定義メソッドの呼び出しポリシーとしてコピーが使用される。
keywords	*fn* の引数長を超えることがないことを指定する *keyword-expression* の結果。	結果の多重定義メソッドの呼び出しポリシーとしてコピーが使用される。

例

```
#include <boost/python/def.hpp>
#include <boost/python/module.hpp>
#include <boost/python/args.hpp>

using namespace boost::python;

char const* foo(int x, int y) { return "foo"; }

BOOST_PYTHON_MODULE(def_test)
{
    def("foo", foo, args("x", "y"), "foo のドキュメンテーション文字列");
}
```

ヘッダ<boost/python/def_visitor.hpp>

はじめに

<boost/python/def_visitor.hpp>は、class_インターフェイスを分散させないようclass_のdefメンバの機能を非侵入的に拡張する汎用的な訪問インターフェイスを提供する。

クラス

def_visitor<DerivedVisitor>クラステンプレート

def_visitorクラスは、その派生クラスを引数にとる（基底）クラスである。def_visitorはプロトコルクラスであり、派生クラスである*DerivedVisitor*はメンバ関数visitを持たなければならない。def_visitorクラスが直接インスタンス化されることはなく、代わりに派生クラス*DerivedVisitor*のインスタンスがclass_のdefメンバ関数の引数として渡される。

def_visitor クラスの概要

```
namespace boost { namespace python {

    template <class DerivedVisitor>
    class def_visitor {};
}
```

def_visitor の要件

クライアントが与える*DerivedVisitor*テンプレート引数は、

- def_visitorから非公開派生していなければならない
- def_visitor_accessクラスへのフレンドアクセスを認めなければらない
- 下表に挙げるvisitメンバ関数のいずれかあるいは両方を定義しなければらない

式	戻り値の型	要件	効果
visitor.visit(cls)	void	*cls*はPythonへラップするclass_のインスタンス。*visitor*はdef_visitorの派生クラス。	cls.def(visitor)の呼び出しがこのメンバ関数へ転送される。
visitor.visit(cls, name, options)	void	*cls*はclass_のインスタンス、*name*はC文字列。*visitor*はdef_visitorの派生クラス。*options*は文脈固有のオプション引数。	cls.def(name, visitor)またはcls.def(name, visitor, options)の呼び出しがこのメンバ関数へ転送される。

例

```
class X {/*...*/};

class my_def_visitor : boost::python::def_visitor<my_def_visitor>
{
    friend class def_visitor_access;

    template <class classT>
    void visit(classT& c) const
    {
        c
            .def("foo", &my_def_visitor::foo)
            .def("bar", &my_def_visitor::bar)
        ;
    }

    static void foo(X& self);
    static void bar(X& self);
};

BOOST_PYTHON_MODULE(my_ext)
{
    class_<X>("X")
        .def(my_def_visitor())
    ;
}
```

ヘッダ<boost/python/docstring_options.hpp>

はじめに

Boost.Python はユーザ定義ドキュメンテーション文字列をサポートし、自動的に C++シグニチャを追加する。これらの機能は既定で有効である。docstring_options クラスはユーザ定義ドキュメンテーション文字列とシグニチャを選択的にまたは両方を抑止できる。

クラス

docstring_options クラス

ラップした関数およびメンバ関数のドキュメンテーション文字列の可視性をインスタンスの寿命に対して制御する。予期しない副作用を防ぐため、インスタンスはコピー不可能である。

docstring_options クラスの概要

```
namespace boost { namespace python {

    class docstring_options : boost::noncopyable
    {
```

```
      public:
          docstring_options(bool show_all=true);

          docstring_options(bool show_user_defined, bool show_signatures);

          docstring_options(bool show_user_defined, bool show_py_signatures, bool
show_cpp_signatures);

          ~docstring_options();

          void
          disable_user_defined();

          void
          enable_user_defined();

          void
          disable_signatures();

          void
          enable_signatures();

          void
          disable_py_signatures();

          void
          enable_py_signatures();

          void
          disable_cpp_signatures();

          void
          enable_cpp_signatures();

          void
          disable_all();

          void
          enable_all();
    };

}}
```

docstring_options クラスのコンストラクタ

```
docstring_options(bool show_all=true);
```

効果

後続のコードで定義する関数およびメンバ関数のドキュメンテーション文字列の可視性を制御する `docstring_options` オブジェクトを構築する。*show_all* が `true` の場合、ユーザ定義のドキュメンテーション文字列と自動的に生成された Python および C++ シグニチャの両方が表示される。*show_all* が `false` の場合、`__doc__` 属性は `None` である。

```
docstring_options(bool show_user_defined, bool show_signatures);
```

効果

後続のコードで定義する関数およびメンバ関数のドキュメンテーション文字列の可視性を制御する docstring_options オブジェクトを構築する。*show_user_defined* が true の場合、ユーザ定義ドキュメンテーション文字列が表示される。*show_signatures* が true の場合、Python および C++のシグニチャが自動的に追加される。*show_user_defined* および *show_signatures* の両方が false の場合、__doc__属性は None である。

```
docstring_options(bool show_user_defined, bool show_py_signatures, bool show_cpp_signatures);
```

効果

後続のコードで定義する関数およびメンバ関数のドキュメンテーション文字列の可視性を制御する docstring_options オブジェクトを構築する。*show_user_defined* が true の場合、ユーザ定義ドキュメンテーション文字列が表示される。*show_py_signatures* が true の場合、Python のシグニチャが自動的に追加される。*show_cpp_signatures* が true の場合、C++のシグニチャが自動的に追加される。すべての引数が false の場合、__doc__属性は None である。

docstring_options クラスのデストラクタ

```
~docstring_options();
```

効果

ドキュメンテーション文字列のオプションを前の状態に復元する。特に docstring_options インスタンスが入れ子の C++スコープ内にある場合は、そのスコープ内の設定が復元される。最後の docstring_options インスタンスがスコープから外れると、既定の「すべて ON」の設定が復元される。

docstring_options クラスの変更関数

```
void disable_user_defined();
void enable_user_defined();
void disable_signatures();
void enable_signatures();
void disable_py_signatures();
void enable_py_signatures();
void disable_cpp_signatures();
void enable_cpp_signatures();
void disable_all();
void enable_all();
```

これらのメンバ関数は、後続のコードのドキュメンテーション文字列の可視性を動的に変更する。*_user_defined() および *_signatures() メンバ関数は細かい制御目的で提供されている。*_all() メンバ関数はすべての設定を同時に操作するための便利なショートカットである。

例

コンパイル時に定義したドキュメンテーション文字列のオプション

```
#include <boost/python/module.hpp>
#include <boost/python/def.hpp>
#include <boost/python/docstring_options.hpp>

void foo() {}

BOOST_PYTHON_MODULE(demo)
{
    using namespace boost::python;
    docstring_options doc_options(DEMO_DOCSTRING_SHOW_ALL);
    def("foo", foo, "foo のドキュメント");
}
```

-DDEMO_DOCSTRING_SHOW_ALL=true としてコンパイルすると次のようになる。

```
>>> import demo
>>> print demo.foo.__doc__
foo() -> None : foo のドキュメント
C++ signature:
    foo(void) -> void
```

-DDEMO_DOCSTRING_SHOW_ALL=false としてコンパイルすると次のようになる。

```
>>> import demo
>>> print demo.foo.__doc__
None
```

選択的な抑止

```
#include <boost/python/module.hpp>
#include <boost/python/def.hpp>
#include <boost/python/args.hpp>
#include <boost/python/docstring_options.hpp>

int foo1(int i) { return i; }
int foo2(long l) { return static_cast<int>(l); }
int foo3(float f) { return static_cast<int>(f); }
int foo4(double d) { return static_cast<int>(d); }

BOOST_PYTHON_MODULE(demo)
{
    using namespace boost::python;
    docstring_options doc_options;
    def("foo1", foo1, arg("i"), "foo1 のドキュメント");
    doc_options.disable_user_defined();
    def("foo2", foo2, arg("l"), "foo2 のドキュメント");
```

```
    doc_options.disable_signatures();
    def("foo3", foo3, arg("f"), "foo3のドキュメント");
    doc_options.enable_user_defined();
    def("foo4", foo4, arg("d"), "foo4のドキュメント");
    doc_options.enable_py_signatures();
    def("foo5", foo4, arg("d"), "foo5のドキュメント");
    doc_options.disable_py_signatures();
    doc_options.enable_cpp_signatures();
    def("foo6", foo4, arg("d"), "foo6のドキュメント");
}
```

Python のコード:

```
>>> import demo
>>> print demo.foo1.__doc__
foo1( (int)i) -> int : foo1のドキュメント
C++ signature:
    foo1(int i) -> int
>>> print demo.foo2.__doc__
foo2( (int)l) -> int :
C++ signature:
    foo2(long l) -> int
>>> print demo.foo3.__doc__
None
>>> print demo.foo4.__doc__
foo4のドキュメント
>>> print demo.foo5.__doc__
foo5( (float)d) -> int : foo5のドキュメント
>>> print demo.foo6.__doc__
foo6のドキュメント
C++ signature:
    foo6(double d) -> int
```

複数の C++スコープからのラッピング

```
#include <boost/python/module.hpp>
#include <boost/python/def.hpp>
#include <boost/python/args.hpp>
#include <boost/python/docstring_options.hpp>

int foo1(int i) { return i; }
int foo2(long l) { return static_cast<int>(l); }

int bar1(int i) { return i; }
int bar2(long l) { return static_cast<int>(l); }

namespace {

    void wrap_foos()
    {
        using namespace boost::python;
        // docstring_optionsを使用していない
        //   -> 外側のC++スコープの設定が適用される
```

```
        def("foo1", foo1, arg("i"), "foo1 のドキュメント");
        def("foo2", foo2, arg("l"), "foo2 のドキュメント");
    }

    void wrap_bars()
    {
        using namespace boost::python;
        bool show_user_defined = true;
        bool show_signatures = false;
        docstring_options doc_options(show_user_defined, show_signatures);
        def("bar1", bar1, arg("i"), "bar1 のドキュメント");
        def("bar2", bar2, arg("l"), "bar2 のドキュメント");
    }
}

BOOST_PYTHON_MODULE(demo)
{
    boost::python::docstring_options doc_options(false);
    wrap_foos();
    wrap_bars();
}
```

Python のコード:

```
>>> import demo
>>> print demo.foo1.__doc__
None
>>> print demo.foo2.__doc__
None
>>> print demo.bar1.__doc__
bar1 のドキュメント
>>> print demo.bar2.__doc__
bar2 のドキュメント
```

boost/libs/python/test/docstring.cpp および docstring.py も見よ。

ヘッダ<boost/python/enum.hpp>

はじめに

<boost/python/enum.hpp>は、ユーザが C++列挙型を Python へエクスポートするためのインターフェイスを定義する。エクスポートする列挙型を引数に持つ enum_クラステンプレートを宣言する。

クラス

enum_<T>クラステンプレート

第 1 引数に渡した C++型に対応する、Python の int 型から派生した Python クラスを作成する。

enum_クラステンプレートの概要

```
namespace boost { namespace python
{
  template <class T>
  class enum_ : public object
  {
    enum_(char const* name, char const* doc = 0);
    enum_<T>& value(char const* name, T);
    enum_<T>& export_values();
  };
}}
```

enum_クラステンプレートのコンストラクタ

```
enum_(char const* name, char const* doc=0);
```

要件

name は Python の識別子の名前付け規約にしたがった ntbs。

効果

名前 *name* の `int` から派生した Python 拡張型を保持する `enum_` オブジェクトを構築する。現在のスコープの名前 *name* の属性を新しい列挙型に束縛する。

enum_クラステンプレートの変更関数

```
inline enum_<T>& value(char const* name, T x);
```

要件

name は Python の識別子の名前付け規約にしたがった ntbs。

効果

ラップした列挙型のインスタンスを名前 *name*、値 *x* とともに型の辞書へ追加する。

```
inline enum_<T>& export_values();
```

効果

`value()` の呼び出しでエクスポートしたすべての列挙値を同じ名前で現在のスコープの属性として設定する。

戻り値

`*this`

例

C++のモジュール定義:

```
#include <boost/python/enum.hpp>
#include <boost/python/def.hpp>
#include <boost/python/module.hpp>

using namespace boost::python;

enum color { red = 1, green = 2, blue = 4 };

color identity_(color x) { return x; }

BOOST_PYTHON_MODULE(enums)
{
    enum_<color>("color")
        .value("red", red)
        .value("green", green)
        .export_values()
        .value("blue", blue)
        ;

    def("identity", identity_);
}
```

Python の対話コード:

```
>>> from enums import *

>>> identity(red)
enums.color.red

>>> identity(color.red)
enums.color.red

>>> identity(green)
enums.color.green

>>> identity(color.green)
enums.color.green

>>> identity(blue)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name blue' is not defined

>>> identity(color.blue)
enums.color.blue

>>> identity(color(1))
enums.color.red

>>> identity(color(2))
enums.color.green

>>> identity(color(3))
enums.color(3)

>>> identity(color(4))
enums.color.blue

>>> identity(1)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: bad argument type for built-in operation
```

ヘッダ<boost/python/errors.hpp>

はじめに

<boost/python/errors.hpp>は、Python と C++の間で例外を管理、変換するための型と関数を提供する。これは Boost.Python でほぼ内部的にのみ使用している、比較的低水準な機能である。ユーザにはほぼ必要ない。

クラス

error_already_set クラス

error_already_set は、Python のエラーが発生したことを示すのに投げられる例外型である。これが投げられた場合、その事前条件は PyErr_Occurred()が true に変換可能な値を返すことである。移植可能なコードではこの例外型を直接投げるべきではなく、代わりに以下の throw_error_already_set()を使用すべきである。

error_already_set クラスの概要

```
namespace boost { namespace python
{
    class error_already_set {};
}}
```

関数

handle_exception

```
template <class T> bool handle_exception(T f) throw();

void handle_exception() throw();
```

要件

第 1 の形式では、式 function0<void>(f)が合法でなければならない。第 2 の形式では、C++例外が現在処理中（C++標準の 15.1 節を見よ）でなければならない。

効果

第 1 の形式では、try ブロック内で f()を呼び出す。最初にすべての登録済みの例外変換器を試す。それらの中で例外を変換できるものがなければ、catch 節で捕捉した C++例外に対する適切な Python 例外を設定し、例外が投げられた場合は true を、

そうでなければ false を返す。第 2 の形式は、現在処理中の例外を再スローする関数を第 1 形式に渡す。

事後条件

処理中の例外が 1 つもない

例外

なし

根拠

言語間の境界においては、C++例外を見逃さないようにすることが重要である。大抵の場合、呼び出し側の言語はスタックを正しい方法で巻き戻す機能を持っていないからである。C++コードを Python API から直接呼び出すときは、例外変換の管理に常に handle_exception を使用せよ。大体の関数ラッピング機能（make_function()、make_constructor()、def() および class_::def()）は自動的にこれを行う。第 2 形式はより便利（以下の例を見よ）だが、内側の try ブロックから例外を再スローした場合に問題を起こすコンパイラが多数ある。

expect_non_null

```
template <class T> T* expect_non_null(T* x);
```

戻り値

x

例外

x == 0 の場合、error_already_set()

根拠

エラー発生時に 0 を返す Python/C API 関数を呼び出すときのエラー処理を容易にする。

throw_error_already_set

```
void throw_error_already_set();
```

効果

throw error_already_set();

根拠

多くのプラットフォームおよびコンパイラでは、共有ライブラリの境界をまたいで投げられた例外を捕捉する首尾一貫した方法がない。Boost.Python ライブラリのこの関数を使用することで、handle_exception() 内で適切な catch ブロックが例外を捕捉できる。

例

```
#include <string>
#include <boost/python/errors.hpp>
#include <boost/python/object.hpp>
#include <boost/python/handle.hpp>
```

```
// obj の"__name__"属性と同じ値を持つ std::string を返す。
std::string get_name(boost::python::object obj)
{
   // __name__属性がなければ例外を投げる
   PyObject* p = boost::python::expect_non_null(
      PyObject_GetAttrString(obj.ptr(), "__name__"));

   char const* s = PyString_AsString(p);
   if (s != 0)
        Py_DECREF(p);

   // Python の文字列でなければ例外を投げる
   std::string result(
      boost::python::expect_non_null(
         PyString_AsString(p)));

   Py_DECREF(p); // p の後始末

   return result;
}

//
// handle_exception の第 1 形式のデモンストレーション
//

// a と b が同じ"__name__"属性を持つ場合は 1、それ以外は 0 の Python Int オブジェクトを
// result に書き込む。
void same_name_impl(PyObject*& result, boost::python::object a, boost::python::object b)
{
   result = PyInt_FromLong(
      get_name(a) == get_name(a2));
}

object borrowed_object(PyObject* p)
{
   return boost::python::object(
        boost::python::handle<>(
             boost::python::borrowed(a1)));
}

// Python 'C' API インターフェイス関数の例
extern "C" PyObject*
same_name(PyObject* args, PyObject* keywords)
{
   PyObject* a1;
   PyObject* a2;
   PyObject* result = 0;

   if (!PyArg_ParseTuple(args, const_cast<char*>("OO"), &a1, &a2))
      return 0;

   // boost::bind を使用してオブジェクトを boost::Function0<void>互換にする
   if (boost::python::handle_exception(
        boost::bind<void>(same_name_impl, boost::ref(result), borrowed_object(a1),
borrowed_object(a2))))
   {
      // 例外が投げられた（Python のエラーが handle_exception()により設定された）
      return 0;
```

```
    }

    return result;
}

//
// handle_exception の第２形式のデモンストレーション。すべてのコンパイラで
// サポートされているわけではない。
//
extern "C" PyObject*
same_name2(PyObject* args, PyObject* keywords)
{
    PyObject* a1;
    PyObject* a2;
    PyObject* result = 0;

    if (!PyArg_ParseTuple(args, const_cast<char*>("OO"), &a1, &a2))
        return 0;

    try {
        return PyInt_FromLong(
            get_name(borrowed_object(a1)) == get_name(borrowed_object(a2)));
    }
    catch(...)
    {
        // 例外が投げられたら、Python へ変換する
        boost::python::handle_exception();
        return 0;
    }
}
```

ヘッダ<boost/python/exception_translator.hpp>

はじめに

他節で述べたように、C++コードが投げた例外が Python のインタープリタコアに渡らないようにすることが重要である。既定では、Boost.Python はラップした関数およびモジュール初期化関数が投げたすべての例外を Python へ変換するが、既定の変換器はきわめて限定的なものである（大半の C++例外は、Python においては`'Unidentifiable C++ Exception'`という表現を持つ RuntimeError 例外である）。以下に述べる例外変換器の登録を行うと、より優れたエラーメッセージを生成できる。

関数

register_exception_translator

```
template<class ExceptionType, class Translate>
void register_exception_translator(Translate translate);
```

要件

Translate は CopyConstructible、かつ以下のコードが合法。

```
void f(ExceptionType x) { translate(x); }
```

式 translate(x) は C++例外を投げるか、後続の PyErr_Occurred() 呼び出しが 1 を返さなければならない。

効果

translate のコピーを例外変換器の列に追加する。この列は Python のコアインタープリタへ渡そうとしている例外を Boost.Python が捕捉したときに試行するものである。新しい変換器は、上で見た catch 節群にマッチするすべての例外を変換するときに最初に呼び出される。後で登録した例外変換器は、より前の例外を変換してもよい。与えられた C++例外を変換できない変換器は再スローしてもよく、そのような例外はより前に登録した変換器（または既定の変換器）が処理する。

例

```
#include <boost/python/module.hpp>
#include <boost/python/def.hpp>
#include <boost/python/exception_translator.hpp>
#include <exception>

struct my_exception : std::exception
{
  char const* what() throw() { return "どれかの例外"; }
};

void translate(my_exception const& e)
{
    // Python 'C' API を使用して例外オブジェクトをセットアップする
    PyErr_SetString(PyExc_RuntimeError, e.what());
}

void something_which_throws()
{
    ...
    throw my_exception();
    ...
}

BOOST_PYTHON_MODULE(exception_translator_ext)
{
  using namespace boost::python;
  register_exception_translator<my_exception>(&translate);

  def("something_which_throws", something_which_throws);
}
```

ヘッダ<boost/python/init.hpp>

はじめに

<boost/python/init.hpp>は、C++コンストラクタを Python へ拡張クラスの__init__関数としてエクスポートするインターフェイスを定義する。

init-expression

init-expression は、拡張クラスについて生成される__init__メソッド群を記述するのに使用する。結果は以下のプロパティを持つ。

docstring	モジュールの__doc__属性に束縛する値を持つ ntbs。
keywords	生成される__init__関数の引数（の残りの部分列）に名前をつけるのに使用する *keyword-expression*。
call policies	CallPolicies モデルのインスタンス。
argument types	ラップした C++オブジェクトを構築するのに使用する C++引数型の MPL 列。*init-expression* は、引数型の接頭辞列が与える**合法な接頭辞**を 1 つ以上持つ。

クラス

init<T1 = unspecified, T2 = unspecified, ...Tn = unspecified>クラステンプレート

1 つ以上の__init__関数群を指定するのに使用する MPL 列。末尾の T*i* のみ optional<...>のインスタンスであってもよい。

init クラステンプレートの概要

```
namespace boost { namespace python
{
  template <T1 = unspecified,...Tn = unspecified>
  struct init
  {
      init(char const* doc = 0);
      template <class Keywords> init(Keywords const& kw, char const* doc = 0);
      template <class Keywords> init(char const* doc, Keywords const& kw);

      template <class CallPolicies>
      unspecified operator[](CallPolicies const& policies) const
  };
}}
```

init クラステンプレートのコンストラクタ

```
init(char const* doc = 0);
template <class Keywords> init(Keywords const& kw, char const* doc = 0);
template <class Keywords> init(char const* doc, Keywords const& kw);
```

要件

doc は与えられた場合 ntbs。*kw* は与えられた場合 *keyword-expression* の結果でなければならない。

効果

結果は、docstring が *doc*、keywords が *kw* への参照である *init-expression* である。第 1 形式を使用した場合、結果の式の keywords は空。式の call policies は default_call_policies のインスタンス。T*n* が optional<U1, U2, ...Um>である場合、式の合法な接頭辞群は次のとおり与えられる。

(T1, T2,...Tn-1), (T1, T2,...Tn-1 , U1), (T1, T2,...Tn-1 , U1, U2), ...(T1, T2,...Tn-1 , U1, U2,...Um)

それ以外の場合、式の**合法な接頭辞**はユーザが指定したテンプレート引数で与えたものとなる。

init クラスのオブザーバ関数

```
template <class Policies>
unspecified operator[](Policies const& policies) const
```

要件

Policies は CallPolicies のモデル。

効果

init オブジェクトとすべて同じプロパティを持ち、call policies のみ *policies* への参照である新しい *init-expression* を返す。

optional<T1 = unspecified, T2 = unspecified, ...Tn = unspecified>クラステンプレート

__init__関数の省略可能引数を指定するのに使用する MPL 列。

optioanal クラステンプレートの概要

```
namespace boost { namespace python
{
  template <T1 = unspecified,...Tn = unspecified>
  struct optional {};
}}
```

例

次の C++宣言があるとすると、

```
class Y;
class X
{
 public:
   X(int x, Y* y) : m_y(y) {}
   X(double);
 private:
   Y* m_y;
};
```

以下のように対応する Boost.Python 拡張クラスを作成できる。

```
using namespace boost::python;

class_<X>("X", "X のドキュメンテーション文字列。",
          init<int,char const*>(args("x","y"), "X.__init__のドキュメンテーション文字列")[
                with_custodian_and_ward<1,3>()]
```

```
        )
    .def(init<double>())
    ;
```

ヘッダ<boost/python/iterator.hpp>

はじめに

<boost/python/iterator.hpp>は、C++のコンテナとイテレータからPython のイテレータを作成するための型と関数を提供する。ある class_がランダムアクセスイテレータをサポートする場合、この機能を使用するより __getitem__（シーケンスプロトコルともいう）を実装したほうがよい。Python は自動的にイテレータ型を作成し（iter()を見よ）、各アクセスは範囲チェックが行われ、不正な C++イテレータを介してアクセスする可能性もない。

クラス

iterator クラステンプレート

iterator<C, P>のインスタンスは、呼び出し可能な Python オブジェクトへの参照を保持する。このオブジェクトは Python から呼び出され、C へ変換可能な単一の引数 c を受け取り、[c.begin(),c.end())を走査する Python イテレータを作成する。省略可能な CallPolicies である P は、走査中に要素をどのように返すか制御するのに使用する。

以下の表において、c は Container のインスタンスである。

テンプレート引数	要件	セマンティクス	既定
Container	[c.begin(),c.end())が合法なイテレータ範囲である。	結果はその引数を c に変換し、c.begin()および c.end()を呼び出しイテレータを得る。Container の const 版 begin()および end()関数を呼び出すには const でなければならない。	
NextPolicies	CallPolicies のデフォルトコンストラクト可能なモデル。	結果のイテレータの next()メソッドに適用される。	常に内部的な C++イテレータを逆参照した結果のコピーを作成する、未規定の CallPolicies モデル。

iterator クラステンプレートの概要

```
namespace boost { namespace python
{
  template <class Container
            , class NextPolicies = unspecified>
  struct iterator : object
```

```
  {
      iterator();
  };
}}
```

iterator クラステンプレートのコンストラクタ

```
iterator()
```

効果

基底クラスを以下の結果で初期化する。

```
range<NextPolicies>(&iterators<Container>::begin, &iterators<Container>::end)
```

事後条件

`this->get()`は、上記のとおり Python のイテレータを作成する Python の呼び出し可能オブジェクトを指す。

根拠

ラップする C++クラスが `begin()` および `end()` を持つようなありふれた場合について、イテレータを容易に作成する方法を提供する。

iterators クラステンプレート

引数の `begin()` および `end()` メンバ関数を確実に呼び出す方法を提供するユーティリティクラスである。C++標準ライブラリにおけるコンテナについて、メンバ関数のアドレスを取る移植可能な方法はないので、それらをラップする場合に `iterators<>`は特に有効である。

以下の表において、`x` は `C` のインスタンスである。

要求する合法な式	型
`x.begin()`	`C` が `const` な型であれば `C::const_iterator` へ変換可能。そうでなければ `C::iterator` へ変換可能。
`x.end()`	`C` が `const` な型であれば `C::const_iterator` へ変換可能。そうでなければ `C::iterator` へ変換可能。

iterators クラステンプレートの概要

```
namespace boost { namespace python
{
  template <class C>
  struct iterators
  {
      typedef typename C::[const_]iterator iterator;
      static iterator begin(C& x);
      static iterator end(C& x);
  };
}}
```

iterators クラステンプレートの入れ子型

C が const 型の場合、

```
typedef typename C::const_iterator iterator;
```

それ以外の場合、

```
typedef typename C::iterator iterator;
```

iterators クラステンプレートの静的関数

```
static iterator begin(C&);
```

戻り値

x.begin()

```
static iterator end(C&);
```

戻り値

x.end()

関数

```
template <class NextPolicies, class Target, class Accessor1, class Accessor2>
object range(Accessor1 start, Accessor2 finish);

template <class NextPolicies, class Accessor1, class Accessor2>
object range(Accessor1 start, Accessor2 finish);

template <class Accessor1, class Accessor2>
object range(Accessor1 start, Accessor2 finish);
```

要件

NextPolicies は、デフォルトコンストラクト可能な CallPolicies モデル。

効果

第 1 形式は Python の呼び出し可能オブジェクトを作成する。このオブジェクトは呼び出されるとその引数を *Target* オブジェクト x に変換し、[bind(start,_1)(x),bind(finish,_1)(x))を走査する Python のイテレータを作成する。イテレータの next() 関数には *NextPolicies* を適用する。第 2 形式は、以下のように *Accessor1* から *Target* を推論する点以外は第 1 形式と同じである。

1. *Accessor1* が関数型の場合、*Target* はその第 1 引数の型。
2. *Accessor1* がデータメンバポインタ R (T::*) の場合、*Target* は T と同じ。
3. *Accessor1* がメンバ関数ポインタ R (T::*)(*arguments*...) *cv-opt*（*cv-opt* は省略可能な cv-qualifier）の場合、*Target*

はTと同じ。

第3形式は、*NextPolicies* が常に内部的なC++イテレータを逆参照した結果のコピーを作成する CallPolicies の未規定のモデルであること以外は第2形式と同じである。

根拠

boost::bind()を使用することで、関数、メンバ関数、データメンバポインタを通じてC++イテレータへアクセスできる。ラップしたクラス型のシーケンス要素のコピーが高コストな場合、*NextPolicies* のカスタマイズが有効である（例えば return_internal_reference を使用する）。*Accessor1* が関数オブジェクトであるか、対象型の基底クラスが他から推論される場合は、*Target* のカスタマイズが有効である。

例

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

#include <vector>

using namespace boost::python;
BOOST_PYTHON_MODULE(demo)
{
    class_<std::vector<double> >("dvec")
        .def("__iter__", iterator<std::vector<double> >())
        ;
}
```

より包括的な例が以下にある。

- libs/python/test/iterator.cpp
- libs/python/test/input_iterator.cpp
- libs/python/test/input_iterator.py

ヘッダ<boost/python/module.hpp>

はじめに

このヘッダは、Boost.Python 拡張モジュールを作成するのに必要な基本的な機能を提供する。

マクロ

BOOST_PYTHON_MODULE(name)は Python のモジュール初期化関数を宣言するのに使用する。引数 *name* は初期化するモジュールの名前に完全に一致していなければならず、Python の識別子の名前付け規約に従っていなければならない。通常、次のように書くところを、

```
extern "C" void initname()
{
   ...
}
```

Boost.Python モジュールでは次のように初期化する。

```
BOOST_PYTHON_MODULE(name)
{
   ...
}
```

このマクロは、使用したスコープ内で 2 つの関数 `extern "C" void init`*name*`()` および `void init_module_`*name*`()` を生成する。関数の本体はマクロ呼び出しの直後でなければならない。生成された C++例外を安全に処理するため、`init_`*name* は `init_module_`*name* を `handle_exception()` に渡す。`init_`*name* の本体内では現在の `scope` は初期化するモジュールを指す。

例

C++のモジュール定義:

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

BOOST_PYTHON_MODULE(xxx)
{
    throw "何かまずいことが起きた"
}
```

Python の対話例:

```
>>> import xxx
Traceback (most recent call last):
  File "", line 1, in ?
RuntimeError: Unidentifiable C++ Exception
```

ヘッダ<boost/python/operators.hpp>

はじめに

`<boost/python/operators.hpp>`は、Python の特殊メソッドを C++の対応する構造から自動的に生成する型と関数を提供する。これらの構造の大半は演算子式であり、ヘッダの名前はそこから来ている。この機能を使用するには、エクスポートする式中でラップするクラス型のオブジェクトを `self` オブジェクトで置き換え、その結果を `class_<>::def()` へ渡す。このヘッダがエクスポートするものの大半は実装部分と考えるべきであり、本稿では詳細について記述しない。

クラス

self_ns::self_t クラス

self_ns::self_t は self オブジェクトの実際の型である。ライブラリは self_t を自身の名前空間 self_ns 内に分離し、他の文脈で引数依存の探索により一般化された演算子テンプレートが発見されるのを防ぐ。ユーザが直接 self_t に触れることはないため、これは実装の詳細と考えるべきである。

self_ns::self_t クラスの概要

```
namespace boost { namespace python { namespace self_ns {
{
   unspecified-type-declaration self_t;

   // 複合演算子
   template <class T> operator_<unspecified> operator+=(self_t, T);
   template <class T> operator_<unspecified> operator-=(self_t, T);
   template <class T> operator_<unspecified> operator*=(self_t, T);
   template <class T> operator_<unspecified> operator/=(self_t, T);
   template <class T> operator_<unspecified> operator%=(self_t, T);
   template <class T> operator_<unspecified> operator>>=(self_t, T);
   template <class T> operator_<unspecified> operator<<=(self_t, T);
   template <class T> operator_<unspecified> operator&=(self_t, T);
   template <class T> operator_<unspecified> operator^=(self_t, T);
   template <class T> operator_<unspecified> operator|=(self_t, T);

   // 比較演算子
   template <class L, class R> operator_<unspecified> operator==(L const&, R const&);
   template <class L, class R> operator_<unspecified> operator!=(L const&, R const&);
   template <class L, class R> operator_<unspecified> operator<(L const&, R const&);
   template <class L, class R> operator_<unspecified> operator>(L const&, R const&);
   template <class L, class R> operator_<unspecified> operator<=(L const&, R const&);
   template <class L, class R> operator_<unspecified> operator>=(L const&, R const&);

   // 非メンバ演算子
   template <class L, class R> operator_<unspecified> operator+(L const&, R const&);
   template <class L, class R> operator_<unspecified> operator-(L const&, R const&);
   template <class L, class R> operator_<unspecified> operator*(L const&, R const&);
   template <class L, class R> operator_<unspecified> operator/(L const&, R const&);
   template <class L, class R> operator_<unspecified> operator%(L const&, R const&);
   template <class L, class R> operator_<unspecified> operator>>(L const&, R const&);
   template <class L, class R> operator_<unspecified> operator<<(L const&, R const&);
   template <class L, class R> operator_<unspecified> operator&(L const&, R const&);
   template <class L, class R> operator_<unspecified> operator^(L const&, R const&);
   template <class L, class R> operator_<unspecified> operator|(L const&, R const&);
   template <class L, class R> operator_<unspecified> pow(L const&, R const&);

   // 単項演算子
   operator_<unspecified> operator-(self_t);
   operator_<unspecified> operator+(self_t);
   operator_<unspecified> operator~(self_t);
   operator_<unspecified> operator!(self_t);

   // 値操作
```

```
    operator_<unspecified> int_(self_t);
    operator_<unspecified> long_(self_t);
    operator_<unspecified> float_(self_t);
    operator_<unspecified> complex_(self_t);
    operator_<unspecified> str(self_t);

    operator_<unspecified> repr(self_t);

}}};
```

各式の結果を class_<>::def() の引数として渡したときに生成されるメソッドを以下の表に挙げる。x はラップするクラス型のオブジェクトである。

self_t クラスの複合演算子

下表において r が other<T>型のオブジェクトの場合、y は型 T のオブジェクトである。それ以外の場合、y は r と同じ型のオブジェクトである。

C++の式	Python のメソッド名	C++の実装
self += r	__iadd__	x += y
self -= r	__isub__	x -= y
self *= r	__imul__	x *= y
self /= r	__idiv__	x /= y
self %= r	__imod__	x %= y
self >>= r	__irshift__	x >>= y
self <<= r	__ilshift__	x <<= y
self &= r	__iand__	x &= y
self ^= r	__ixor__	x ^= y
self |= r	__ior__	x |= y

self_t クラスの比較関数

下表において r が self_t 型の場合、y は x と同じ型のオブジェクトである。l か r が other<T>型のオブジェクトの場合、y は型 T のオブジェクトである。それ以外の場合、y は l か r と同じ型のオブジェクトであり、l は self_t 型以外である。

「Python の式」の列は、x および y の型へ変換可能なオブジェクトについて Python がサポートする式を表す。2 番目の演算は Python の高水準比較演算子の反射則（可換則）により生じるもので、対応する演算が y オブジェクトのメソッドとして定義されない場合のみ使用される。

C++の式	Python のメソッド名	C++の実装	Python の式（1 番目、2 番目）
self == r	__eq__	x == y	x == y, y == x
l == self	__eq__	y == x	y == x, x == y
self != r	__ne__	x != y	x != y, y != x
l != self	__ne__	y != x	y != x, x != y
self < r	__lt__	x < y	x < y, y > x
l < self	__gt__	y < x	y > x, x < y

self > r	__gt__	x > y	x > y, y < x
l > self	__lt__	y > x	y < x, x > y
self <= r	__le__	x <= y	x <= y, y >= x
l <= self	__ge__	y <= x	y >= x, x <= y
self >= r	__ge__	x >= y	x >= y, y <= x
l >= self	__le__	y >= x	y <= x, x >= y

self_t クラスの非メンバ演算子

ここで述べられているように、以下の名前が「__r」で始まる演算は左オペランドが与えられた演算をサポートしない場合のみ呼び出される。

C++の式	Python のメソッド名	C++の実装
self + r	__add__	x + y
l + self	__radd__	y + x
self - r	__sub__	x - y
l - self	__rsub__	y - x
self * r	__mul__	x * y
l * self	__rmul__	y * x
self / r	__div__	x / y
l / self	__rdiv__	y / x
self % r	__mod__	x % y
l % self	__rmod__	y % x
self >> r	__rshift__	x >> y
l >> self	__rrshift__	y >> x
self << r	__lshift__	x << y
l << self	__rlshift__	y << x
self & r	__and__	x & y
l & self	__rand__	y & x
self ^ r	__xor__	x ^ y
l ^ self	__rxor__	y ^ x
self | r	__or__	x | y
l | self	__ror__	y | x
pow(self, r)	__pow__	pow(x, y)
pow(l, self)	__rpow__	pow(y, x)

self_t クラスの単項演算子

C++の式	Python のメソッド名	C++の実装
-self	__neg__	-x
+self	__pos__	+x
~self	__invert__	~x
not self	__nonzero__	!!x

C++の式	Python のメソッド名	C++の実装
または !self		

self_t クラスの値演算

C++の式	Python のメソッド名	C++の実装
int_(self)	__int__	long(x)
long_	__long__	PyLong_FromLong(x)
float_	__float__	double(x)
complex_	__complex__	std::complex<double>(x)
str	__str__	lexical_cast<std::string>(x)
repr	__repr__	lexical_cast<std::string>(x)

other クラステンプレート

other<T>のインスタンスは self とともに演算子式中で使用し、結果は同じ式の other<T>を T オブジェクトで置き換えたものと等価である。T オブジェクトの構築が高価で避けたい場合、コンストラクタが利用できない場合、または単純にコードを明確にする場合に other<T>を使用するとよい。

other クラステンプレートの概要

```
namespace boost { namespace python
  template <class T>
  struct other
  {
  };
}}
```

detail::operator_クラステンプレート

detail::operator_<>のインスタンス化は、self を含む演算子式の戻り値型として使用する。これは実装の詳細として考えるべきであり、self 式の結果がどのように class_<>::def()呼び出しとマッチするか見るためとしてのみ、ここに記載する。

detail::operator_クラステンプレートの概要

```
namespace boost { namespace python { namespace detail
{
  template <unspecified>
  struct operator_
  {
  };
}}}
```

オブジェクト

self

```
namespace boost { namespace python
{
  using self_ns::self;
}}
```

例

```
#include <boost/python/module.hpp>
#include <boost/python/class.hpp>
#include <boost/python/operators.hpp>
#include <boost/operators.hpp>

struct number
   : boost::integer_arithmetic<number>
{
    explicit number(long x_) : x(x_) {}
    operator long() const { return x; }

    template <class T>
    number& operator+=(T const& rhs)
    { x += rhs; return *this; }

    template <class T>
    number& operator-=(T const& rhs)
    { x -= rhs; return *this; }

    template <class T>
    number& operator*=(T const& rhs)
    { x *= rhs; return *this; }

    template <class T>
    number& operator/=(T const& rhs)
    { x /= rhs; return *this; }

    template <class T>
    number& operator%=(T const& rhs)
    { x %= rhs; return *this; }

   long x;
};

using namespace boost::python;
BOOST_PYTHON_MODULE(demo)
{
   class_<number>("number", init<long>())
      // self との組み合わせ
      .def(self += self)
      .def(self + self)
      .def(self -= self)
      .def(self - self)
      .def(self *= self)
```

```
        .def(self * self)
        .def(self /= self)
        .def(self / self)
        .def(self %= self)
        .def(self % self)

        // Python の int への変換
        .def(int_(self))

        // long との組み合わせ
        .def(self += long())
        .def(self + long())
        .def(long() + self)
        .def(self -= long())
        .def(self - long())
        .def(long() - self)
        .def(self *= long())
        .def(self * long())
        .def(long() * self)
        .def(self /= long())
        .def(self / long())
        .def(long() / self)
        .def(self %= long())
        .def(self % long())
        .def(long() % self)
        ;
}
```

ヘッダ<boost/python/scope.hpp>

はじめに

Python のスコープ（名前空間）を問い合わせたり制御する機能を定義する。このスコープには新しくラップするクラスや関数を追加できる。

クラス

scope クラス

scope クラスは、新しい拡張クラスおよびラップした関数がその属性として定義される Python の名前空間を制御する（自分自身に対応した）グローバルな Python オブジェクトを持つ。新しい scope オブジェクトをデフォルトコンストラクタで構築すると、そのオブジェクトは対応するグローバルな Python のオブジェクトに束縛される。scope オブジェクトを引数付きで構築すると、対応するグローバルな Python のオブジェクトを引数が保持するオブジェクトへ変更する。これはこの scope オブジェクトの寿命が終わるまで続き、その時点で対応するグローバルな Python オブジェクトは scope オブジェクトを構築する前の状態に復元する。

scope クラスの概要

```
namespace boost { namespace python
{
  class scope : public object
  {
   public:
      scope(scope const&);
      scope(object const&);
      scope();
      ~scope()
   private:
      void operator=(scope const&);
  };
}}
```

scope クラスのコンストラクタおよびデストラクタ

```
explicit scope(scope const& x);
explicit scope(object const& x);
```

現在のスコープ相当オブジェクトへの参照を格納し、スコープ相当オブジェクトに `x.ptr()` が参照するオブジェクトを設定する。`object` 基底クラスを *x* で初期化する。

```
scope();
```

現在のスコープ相当オブジェクトへの参照を格納する。`object` 基底クラスを現在のスコープ相当オブジェクトで初期化する。モジュール初期化関数の外部では、現在の相当 Python オブジェクトは `None` である。

```
~scope()
```

現在の相当 Python オブジェクトを格納したオブジェクトに設定する。

例

以下の例は、スコープの設定を使用して入れ子クラスを定義する方法を示している。

C++のモジュール定義:

```
#include <boost/python/module.hpp>
#include <boost/python/class.hpp>
#include <boost/python/scope.hpp>
using namespace boost::python;

struct X
{
  void f() {}
```

```
  struct Y { int g() { return 42; } };
};

BOOST_PYTHON_MODULE(nested)
{
   // 現在の（モジュールの）スコープにいくつか定数を追加する
   scope().attr("yes") = 1;
   scope().attr("no") = 0;

   // 現在のスコープを変更する
   scope outer
       = class_<X>("X")
            .def("f", &X::f)
            ;

   // 現在のスコープ X でクラス Y を定義する
   class_<X::Y>("Y")
      .def("g", &X::Y::g)
      ;
}
```

Python の対話例：

```
>>> import nested
>>> nested.yes
1
>>> y = nested.X.Y()
>>> y.g()
42
```

ヘッダ<boost/python/stl_iterator.hpp>

はじめに

<boost/python/stl_iterator.hpp>は、Python の走査可能オブジェクトから C++のイテレータを作成する型を提供する。

クラス

stl_input_iterator クラステンプレート

stl_input_iterator<T>のインスタンスは Python のイテレータを保持し、それを STL アルゴリズムで使用できるよう適合させる。stl_input_iterator<T>は入力イテレータの要件を満たす。

テンプレート引数	要件	セマンティクス	既定
ValueType	ValueType がコピー構築可能でなければならない。	stl_input_iterator<T>のインスタンスを参照剥がしすると ValueType 型の rvalue が返る。	（なし）

stl_input_iterator クラステンプレートの概要

```
namespace boost { namespace python
{
  template <class ValueType>
  struct stl_input_iterator
  {
      typedef std::ptrdiff_t difference_type;
      typedef ValueType value_type;
      typedef ValueType* pointer;
      typedef ValueType reference;
      typedef std::input_iterator_tag iterator_category;

      stl_input_iterator();
      stl_input_iterator(object const& ob);

      stl_input_iterator& operator++();
      stl_input_iterator operator++(int);

      ValueType operator*() const;

      friend bool operator==(stl_input_iterator const& lhs, stl_input_iterator const& rhs);
      friend bool operator!=(stl_input_iterator const& lhs, stl_input_iterator const& rhs);
  private:
      object it; // 説明のためにのみ記載
      object ob; // 説明のためにのみ記載
  };
}}
```

stl_input_iterator クラステンプレートのコンストラクタ

```
stl_input_iterator()
```

効果

シーケンスの終端を表すのに使用する、末尾の直後を指す入力イテレータを作成する。

事後条件

`this` が末尾の直後。

例外

なし。

```
stl_input_iterator(object const& ob)
```

効果

`ob.attr("__iter__")()` を呼び出し、結果の Python のイテレータオブジェクトを `this->it` に格納する。次に `this->it.attr("next")()` を呼び出し、結果を `this->ob` に格納する。シーケンスに走査するものが残っていない場合は `this->ob` に `object()` を設定する。

事後条件

this が参照剥がし可能か末尾の直後。

stl_input_iterator クラステンプレートの変更メソッド

```
stl_input_iterator& operator++()
```

効果

this->it.attr("next")()を呼び出し、結果を this->ob に格納する。シーケンスに走査するものが残っていない場合は this->ob に object()を設定する。

事後条件

this が参照剥がし可能か末尾の直後。

戻り値

*this。

```
stl_input_iterator operator++(int)
```

効果

stl_input_iterator tmp = *this; ++*this; return tmp;

事後条件

this が逆参照可能か末尾の直後。

stl_input_iterator クラステンプレートのオブザーバ

```
ValueType operator*() const
```

効果

シーケンス内の現在の要素を返す。

戻り値

extract<ValueType>(this->ob);

```
friend bool operator==(stl_input_iterator const& lhs, stl_input_iterator const& rhs)
```

効果

両方のイテレータが参照剥がし可能であるか両方のイテレータが末尾の直後であれば真、それ以外は偽を返す。

戻り値

(lhs.ob == object()) == (rhs.ob == object())

```
friend bool operator!=(stl_input_iterator const& lhs, stl_input_iterator const& rhs)
```

効果

両方のイテレータが参照剥がし可能であるか両方のイテレータが末尾の直後であれば偽、それ以外は真を返す。

戻り値

`!(lhs == rhs)`

例

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

#include <list>

using namespace boost::python;
std::list<int> sequence_to_int_list(object const& ob)
{
    stl_input_iterator<int> begin(ob), end;
    return std::list<int>(begin, end);
}
```

ヘッダ<boost/python/wrapper.hpp>

はじめに

仮想関数を「Python でオーバーライド」可能なクラス `T` をラップするため、つまり C++から仮想関数を呼び出すときに Python における派生クラスの当該メソッドが呼び出されるよう、これらの仮想関数を Python から呼び出せるようオーバーライドする C++ラッパクラスを `T` から派生させて作成しなければならない。このヘッダのクラスを使用して、そういった作業を容易にできる。

クラス

override クラス

C++仮想関数の Python オーバーライドをカプセル化する。`override` オブジェクトは Python の呼び出し可能オブジェクトか `None` のいずれかを保持する。

override クラスの概要

```
namespace boost
{
  class override : object
  {
   public:
      unspecified operator() const;
      template <class A0>
      unspecified operator(A0) const;
      template <class A0, class A1>
      unspecified operator(A0, A1) const;
```

```
      ...
      template <class A0, class A1, ...class An>
      unspecified operator(A0, A1, ...An) const;
  };
};
```

override クラスのオブザーバ関数

```
unspecified operator() const;
template <class A0>
unspecified operator(A0) const;
template <class A0, class A1>
unspecified operator(A0, A1) const;
...
template <class A0, class A1, ...class An>
unspecified operator(A0, A1, ...An) const;
```

効果

`*this` が Python の呼び出し可能オブジェクトを保持している場合、ここに示す方法で指定した引数で呼び出す。それ以外の場合、error_already_set を投げる。

戻り値

Python の呼び出し結果を保持する未規定型のオブジェクト。C++型 `R` への変換は、結果のオブジェクトの `R` への変換により行われる。変換に失敗した場合、error_already_set を投げる。

wrapper クラステンプレート

ラッパクラスを `T` と `wrapper<T>`の両方から派生することで、その派生クラスの記述が容易になる。

wrapper クラステンプレートの概要

```
namespace boost
{
  class wrapper
  {
   protected:
      override get_override(char const* name) const;
  };
};
```

wrapper クラスのオブザーバ関数

```
override get_override(char const* name) const;
```

要件

name は ntbs。

戻り値

Python 派生クラスインスタンスが名前 *name* の関数をオーバーライドしているとして、`*this` がその C++基底クラスのサブオブジェクトである場合、Python のオーバーライドに委譲する `override` オブジェクトを返す。それ以外の場合、`None` を保持する `override` オブジェクトを返す。

例

```
#include <boost/python/module.hpp>
#include <boost/python/class.hpp>
#include <boost/python/wrapper.hpp>
#include <boost/python/call.hpp>

using namespace boost::python;

// 純粋仮想関数を１つ持つクラス
struct P
{
    virtual ~P(){}
    virtual char const* f() = 0;
    char const* g() { return "P::g()"; }
};

struct PCallback : P, wrapper<P>
{
    char const* f()
    {
#if BOOST_WORKAROUND(BOOST_MSVC, <= 1300) // vc6/vc7 のための workaround
        return call<char const*>(this->get_override("f").ptr());
#else
        return this->get_override("f")();
#endif
    }
};

// 非純粋仮想関数を１つ持つクラス
struct A
{
    virtual ~A(){}
    virtual char const* f() { return "A::f()"; }
};

struct ACallback :  A, wrapper<A>
{
    char const* f()
    {
        if (override f = this->get_override("f"))
#if BOOST_WORKAROUND(BOOST_MSVC, <= 1300) // vc6/vc7 のための workaround
            return call<char const*>(f.ptr());
#else
            return f();
#endif

        return A::f();
    }

    char const* default_f() { return this->A::f(); }
```

```
};

BOOST_PYTHON_MODULE_INIT(polymorphism)
{
    class_<PCallback,boost::noncopyable>("P")
        .def("f", pure_virtual(&P::f))
        ;

    class_<ACallback,boost::noncopyable>("A")
        .def("f", &A::f, &Acallback::default_f)
        ;
}
```

オブジェクトラッパ

ヘッダ<boost/python/dict.hpp>

はじめに

Python の dict 型に対する TypeWrapper をエクスポートする。

クラス

dict クラス

Python の組み込み dict 型のマップ型プロトコルをエクスポートする。以下に定義するコンストラクタとメンバ関数のセマンティクスを完全に理解するには、TypeWrapper コンセプトの定義を読むことである。dict は object から公開派生しているので、object の公開インターフェイスは dict のインスタンスにも当てはまる。

dict クラスの概要

```
namespace boost { namespace python
{
   class dict : public object
   {
      dict();

      template< class T >
      dict(T const & data);

      // 変更
      void clear();
      dict copy();

      template <class T1, class T2>
      tuple popitem();

      template <class T>
```

```
      object setdefault(T const &k);

      template <class T1, class T2>
      object setdefault(T1 const & k, T2 const & d);

      void update(object_cref E);

      template< class T >
      void update(T const & E);

      // オブザーバ
      list values() const;

      object get(object_cref k) const;

      template<class T>
      object get(T const & k) const;

      object get(object_cref k, object_cref d) const;
      object get(T1 const & k, T2 const & d) const;

      bool has_key(object_cref k) const;

      template< class T >
      bool has_key(T const & k) const;

      list items() const;
      object iteritems() const;
      object iterkeys() const;
      object itervalues() const;
      list keys() const;
  };
}}
```

例

```
using namespace boost::python;
dict swap_object_dict(object target, dict d)
{
    dict result = extract<dict>(target.attr("__dict__"));
    target.attr("__dict__") = d;
    return result;
}
```

ヘッダ<boost/python/list.hpp>

はじめに

Python の `list` 型に対する TypeWrapper をエクスポートする。

クラス

list クラス

Python の組み込み `list` 型のシーケンス型プロトコルをエクスポートする。以下に定義するコンストラクタとメンバ関数のセマンティクスを完全に理解するには、TypeWrapper コンセプトの定義を読むことである。`list` は `object` から公開派生しているので、`object` の公開インターフェイスは `list` のインスタンスにも当てはまる。

list クラスの概要

```
namespace boost { namespace python
{
  class list : public object
  {
   public:
      list(); // list を新規作成する

      template <class T>
      explicit list(T const& sequence);

      template <class T>
      void append(T const& x);

      template <class T>
      long count(T const& value) const;

      template <class T>
      void extend(T const& x);

      template <class T>
      long index(T const& x) const;

      template <class T>
      void insert(object const& index, T const& x); // index の前にオブジェクトを挿入する

      object pop(); // index 位置（既定は末尾）の要素を削除して返す
      object pop(long index);
      object pop(object const& index);

      template <class T>
      void remove(T const& value);

      void reverse(); //  「その場で」逆順に入れ替える

      void sort(); //  「その場で」ソートする。比較関数を与える場合の形式は cmpfunc(x, y) -> -1, 0, 1

      template <class T>
      void sort(T const& value);
  };
}}
```

例

```
using namespace boost::python;

// リスト内の 0 の数を返す
long zeroes(list l)
{
   return l.count(0);
}
```

ヘッダ<boost/python/long.hpp>

はじめに

Python の long 整数型に対する TypeWrapper をエクスポートする。

クラス

long_クラス

Python の組み込み long 型の数値型プロトコルをエクスポートする。以下に定義するコンストラクタとメンバ関数のセマンティクスを完全に理解するには、TypeWrapper コンセプトの定義を読むことである。long_ は object から公開派生しているので、object の公開インターフェイスは long_ のインスタンスにも当てはまる。

long_クラスの概要

```
namespace boost { namespace python
{
  class long_ : public object
  {
   public:
      long_(); // long_を新規に作成する

      template <class T>
      explicit long_(T const& rhs);

      template <class T, class U>
      long_(T const& rhs, U const& base);
  };
}}
```

例

```
namespace python = boost::python;
```

```
// オーバーフローすることなく階乗を計算する
python::long_ fact(long n)
{
   if (n == 0)
      return python::long_(1);
   else
      return n * fact(n - 1);
}
```

ヘッダ<boost/python/numeric.hpp>

はじめに

Python の array 型に対する TypeWrapper をエクスポートする。

クラス

array クラス

Numerical Python[14]の Numeric および NumArray モジュールの配列型へのアクセスを提供する。後述の関数群を除き、以下に定義するコンストラクタとメンバ関数のセマンティクスを完全に理解するには、TypeWrapper コンセプトの定義を読むことである。array は object から公開派生しているので、object の公開インターフェイスは array のインスタンスにも当てはまる。

numarray モジュールが既定の位置にインストールされている場合、Python の対応する型として numarray.NDArray を既定で使用する。それ以外の場合、Numeric.ArrayType にフォールバックする。いずれの拡張モジュールもインストールされていない場合、ラップした関数を numeric::array 引数で多重定義したものはマッチすることなく、他で numeric::array を使用しようとすると適切な Python の例外を送出する。対応する Python の型は set_module_and_type(...) 静的関数で手動で設定できる。

array クラスの概要

```
namespace boost { namespace python { namespace numeric
{
   class array : public object
   {
    public:
      object astype();
      template <class Type>
      object astype(Type const& type_);

      template <class Type>
      array new_(Type const& type_) const;
```

14 訳注　現在は NumPy というパッケージに変わっています。Numeric モジュールは numpy.oldnumeric、NumArray モジュールは numpy.numarray という名前で互換モジュールが残されています。現在このクラスを使用する場合 NumPy をインストールした状態で後述の array.set_module_and_type 関数を使用して numpy.ndarray クラスを指示するのが一般的ですが、古いクラスとインターフェイスが完全に一致しているわけではありません。

```
      template <class Sequence>
      void resize(Sequence const& x);
      void resize(long x1);
      void resize(long x1, long x2);
      ...
      void resize(long x1, long x2,...long xn);

      template <class Sequence>
      void setshape(Sequence const& x);
      void setshape(long x1);
      void setshape(long x1, long x2);
      ...
      void setshape(long x1, long x2,...long xn);

      template <class Indices, class Values>
      void put(Indices const& indices, Values const& values);

      template <class Sequence>
      object take(Sequence const& sequence, long axis = 0);

      template <class File>
      void tofile(File const& f) const;

      object factory();
      template <class Sequence>
      object factory(Sequence const&);
      template <class Sequence, class Typecode>
      object factory(Sequence const&, Typecode const&, bool copy = true, bool savespace = false);
      template <class Sequence, class Typecode, class Type>
      object factory(Sequence const&, Typecode const&, bool copy, bool savespace, Type const&);
      template <class Sequence, class Typecode, class Type, class Shape>
      object factory(Sequence const&, Typecode const&, bool copy, bool savespace, Type const&,
Shape const&);

      template <class T1>
      explicit array(T1 const& x1);
      template <class T1, class T2>
      explicit array(T1 const& x1, T2 const& x2);
      ...
      template <class T1, class T2,...class Tn>
      explicit array(T1 const& x1, T2 const& x2,...Tn const& xn);

      static void set_module_and_type();
      static void set_module_and_type(char const* package_path = 0, char const* type_name = 0);
      static void get_module_name();

      object argmax(long axis=-1);

      object argmin(long axis=-1);

      object argsort(long axis=-1);

      void byteswap();

      object copy() const;

      object diagonal(long offset = 0, long axis1 = 0, long axis2 = 1) const;

      void info() const;

      bool is_c_array() const;
```

```
      bool isbyteswapped() const;
      void sort();
      object trace(long offset = 0, long axis1 = 0, long axis2 = 1) const;
      object type() const;
      char typecode() const;

      object getflat() const;
      long getrank() const;
      object getshape() const;
      bool isaligned() const;
      bool iscontiguous() const;
      long itemsize() const;
      long nelements() const;
      object nonzero() const;

      void ravel();

      object repeat(object const& repeats, long axis=0);

      void setflat(object const& flat);

      void swapaxes(long axis1, long axis2);

      str tostring() const;

      void transpose(object const& axes = object());

      object view() const;
  };
}}}
```

array クラスのオブザーバ関数

```
object factory();
template <class Sequence>
object factory(Sequence const&);
template <class Sequence, class Typecode>
object factory(Sequence const&, Typecode const&, bool copy = true, bool savespace = false);
template <class Sequence, class Typecode, class Type>
object factory(Sequence const&, Typecode const&, bool copy, bool savespace, Type const&);
template <class Sequence, class Typecode, class Type, class Shape>
object factory(Sequence const&, Typecode const&, bool copy, bool savespace, Type const&, Shape
const&);
```

これらの関数は内部的な配列型の `array()` 関数群にマップする。「`array`」という名前でないのは、メンバ関数をクラスと同じ名前で定義できないというC++の制限による。

```
template <class Type>
array new_(Type const&) const;
```

この関数は内部的な配列型の `new()` 関数にマップする。「`new`」という名前でないのは、C++のキーワードだからである。

array クラスの静的関数

```
static void set_module_and_type(char const* package_path, char const* type_name);
static void set_module_and_type();
```

要件

package_path と *type_name* は、与えられた場合は ntbs。

効果

第 1 形式は、名前 *type_name* の型を与えるモジュールのパッケージパスを *package_path* に設定する。第 2 形式は、既定の探索動作を復元する。対応する Python の型は必要となった最初の 1 回目、および以降の `set_module_and_type` 呼び出し後に必要となった最初の 1 回目のみ探索される。

```
static std::string get_module_name()
```

効果

新しい `numeric::array` インスタンスが保持するクラスを含むモジュールの名前を返す。

例

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

// 二次元配列の最初の要素を設定する
void set_first_element(numeric::array& y, double value)
{
    y[make_tuple(0,0)] = value;
}
```

ヘッダ<boost/python/object.hpp>

はじめに

汎用的な Python のオブジェクトラッパクラス `object` および関連クラスをエクスポートする。引数依存の探索および `object` が定義する汎化演算子に絡む潜在的な問題を避けるため、これらの機能はすべて `namespace boost::python::api` で定義され、`object` は `using` 宣言で `namespace boost::python` へインポートされている。

型

slice_nil

```
class slice_nil;
```

```
static const _ = slice_nil();
```

次のように、Python のスライス式で添字を省く効果を得る型。

```
>>> x[:-1]
>>> x[::-1]
```

C++で等価なことをするには、

```
x.slice(_,-1)
x[slice(_,_,-1)]
```

クラス

const_attribute_policies クラス

`const object` への属性アクセスを表現するプロキシのためのポリシー。

const_attribute_policies クラスの概要

```
namespace boost { namespace python { namespace api
{
  struct const_attribute_policies
  {
      typedef char const* key_type;
      static object get(object const& target, char const* key);
  };
}}}
```

const_attribute_policies クラスの静的関数

```
static object get(object const& target, char const* key);
```

要件

key が ntbs。

効果

target の属性 *key* にアクセスする。

戻り値

属性アクセスの結果を管理する `object`。

例外

Python の例外が送出した場合 `error_already_set`。

attribute_policies クラス

変更可能な `object` への属性アクセスを表現するプロキシのためのポリシー。

attribute_policies クラスの概要

```
namespace boost { namespace python { namespace api
{
  struct attribute_policies : const_attribute_policies
  {
      static object const& set(object const& target, char const* key, object const& value);
      static void del(object const&target, char const* key);
  };
}}}
```

attribute_policies クラスの静的関数

```
static object const& set(object const& target, char const* key, object const& value);
```

要件

key が ntbs。

効果

target の属性 *key* に *value* を設定する。

例外

Python の例外が送出した場合 `error_already_set`。

```
static void del(object const& target, char const* key);
```

要件

key が ntbs。

効果

target の属性 *key* を削除する。

例外

Python の例外が送出した場合 `error_already_set`。

const_objattribute_policies クラス

`const object` へのアクセス属性（属性名を `const object` で与える場合）を表現するプロキシのためのポリシー。

const_objattribute_policies クラスの概要

```
namespace boost { namespace python { namespace api
{
  struct const_objattribute_policies
  {
      typedef object const& key_type;
      static object get(object const& target, object const& key);
  };
}}}
```

const_objattribute_policies クラスの静的関数

```
static object get(object const& target, object const& key);
```

要件

key が文字列を保持する `object`。

効果

target の属性 *key* にアクセスする。

戻り値

属性アクセスの結果を管理する `object`。

例外

Python の例外が送出した場合 `error_already_set`。

objattribute_policies クラス

変更可能な `object` への属性アクセス（属性名を `const object` で与える場合）を表現するプロキシのためのポリシー。

objattribute_policies クラスの概要

```
namespace boost { namespace python { namespace api
{
  struct objattribute_policies : const_objattribute_policies
  {
      static object const& set(object const& target, object const& key, object const& value);
      static void del(object const&target, object const& key);
  };
}}}
```

objattribute_policies クラスの静的関数

```
static object const& set(object const& target, object const& key, object const& value);
```

要件

key が文字列を保持する `object`。

効果

target の属性 *key* に *value* を設定する。

例外

Python の例外が送出した場合 `error_already_set`。

```
static void del(object const&target, object const& key);
```

要件

key が文字列を保持する `object`。

効果

target の属性 *key* を削除する。

例外

Python の例外が送出した場合 `error_already_set`。

const_item_policies クラス

`const object` への（Python の角括弧演算子 `[]` による）要素アクセスを表現するプロキシのためのポリシー。

const_item_policies クラスの概要

```
namespace boost { namespace python { namespace api
{
  struct const_item_policies
  {
      typedef object key_type;
      static object get(object const& target, object const& key);
  };
}}}
```

const_item_policies クラスの静的関数

```
static object get(object const& target, object const& key);
```

効果

target の *key* で指定する要素へアクセスする。

戻り値

要素アクセスの結果を管理する `object`。

例外

Python の例外が送出した場合 `error_already_set`。

item_policies クラス

変更可能な `object` への（Python の角括弧演算子 `[]` による）要素アクセスを表現するプロキシのためのポリシー。

item_policies クラスの概要

```
namespace boost { namespace python { namespace api
{
  struct item_policies : const_item_policies
  {
      static object const& set(object const& target, object const& key, object const& value);
      static void del(object const& target, object const& key);
  };
}}}
```

item_policies クラスの静的関数

```
static object const& set(object const& target, object const& key, object const& value);
```

効果

target の *key* で指定する要素を *value* に設定する。

例外

Python の例外が送出した場合 error_already_set。

```
static void del(object const& target, object const& key);
```

効果

target の *key* で指定する要素を削除する。

例外

Python の例外が送出した場合 error_already_set。

const_slice_policies クラス

`const object` への（Python のスライス表記 $[x:y]$ による）スライスアクセスを表現するプロキシのためのポリシー。

const_slice_policies クラスの概要

```
namespace boost { namespace python { namespace api
{
  struct const_slice_policies
  {
      typedef std::pair<handle<>, handle<> > key_type;
      static object get(object const& target, key_type const& key);
  };
```

```
}}}
```

const_slice_policies クラスの静的関数

```
static object get(object const& target, key_type const& key);
```

効果

target の *key* で指定するスライスへアクセスする。

戻り値

スライスアクセスの結果を管理する `object`。

例外

Python の例外が送出した場合 `error_already_set`。

slice_policies クラス

変更可能 `object` へのスライスアクセスを表現するプロキシのためのポリシー。

slice_policies クラスの概要

```
namespace boost { namespace python { namespace api
{
  struct slice_policies : const_slice_policies
  {
      static object const& set(object const& target, key_type const& key, object const& value);
      static void del(object const& target, key_type const& key);
  };
}}}
```

slice_policies クラスの静的関数

```
static object const& set(object const& target, key_type const& key, object const& value);
```

効果

target の *key* で指定するスライスに *value* を設定する。

例外

Python の例外が送出した場合 `error_already_set`。

```
static void del(object const& target, key_type const& key);
```

効果

target の *key* で指定するスライスを削除する。

例外

Python の例外が送出した場合 error_already_set。

object_operators<U>クラステンプレート

これは object およびその proxy テンプレートの基底クラスであり、共通のインターフェイス（メンバ関数およびクラス本体内で定義しなければならない演算子）を提供する。テンプレート引数 *U* は object_operators<U>の派生型という想定である。実際にはユーザはこのクラスを直接使用すべきではないが、object とそのプロキシに対して重要なインターフェイスを提供するので、ここに記載する。

object_operators クラステンプレートの概要

```
namespace boost { namespace python { namespace api
{
  template <class U>
  class object_operators
  {
   public:
      // 関数呼び出し
      //
      object operator()() const;

      template <class A0>
      object operator()(A0 const&) const;
      template <class A0, class A1>
      object operator()(A0 const&, A1 const&) const;
      ...
      template <class A0, class A1,...class An>
      object operator()(A0 const&, A1 const&,...An const&) const;

      detail::args_proxy operator* () const;
      object operator()(detail::args_proxy const &args) const;
      object operator()(detail::args_proxy const &args,
                        detail::kwds_proxy const &kwds) const;

      // 真偽値のテスト
      //
      typedef unspecified bool_type;
      operator bool_type() const;

      // 属性アクセス
      //
      proxy<const_object_attribute> attr(char const*) const;
      proxy<object_attribute> attr(char const*);
      proxy<const_object_objattribute> attr(object const&) const;
      proxy<object_objattribute> attr(object const&);

      // 要素アクセス
      //
      template <class T>
      proxy<const_object_item> operator[](T const& key) const;

      template <class T>
```

```
      proxy<object_item> operator[](T const& key);

      // スライシング
      //
      template <class T, class V>
      proxy<const_object_slice> slice(T const& start, V const& end) const

      template <class T, class V>
      proxy<object_slice> slice(T const& start, V const& end);
  };
}}}
```

object_operators クラステンプレートのオブザーバ関数

```
object operator()() const;
template <class A0>
object operator()(A0 const&) const;
template <class A0, class A1>
object operator()(A0 const&, A1 const&) const;
...
template <class A0, class A1,...class An>
object operator()(A0 const&, A1 const&,...An const&) const;
```

効果

call<object>(object(*static_cast<U*>(this)).ptr(), a1, a2,...aN)

```
object operator()(detail::args_proxy const &args) const;
```

効果

タプル *args* で与えた引数で `object` を呼び出す。

```
object operator()(detail::args_proxy const &args,
                  detail::kwds_proxy const &kwds) const;
```

効果

タプル *args* で与えた引数と辞書 *kwds* で与えた名前付き引数で `object` を呼び出す。

```
operator bool_type() const;
```

効果

`*this` の真偽値をテストする。

戻り値

call<object>(object(*static_cast<U*>(this)).ptr(), a1, a2,...aN)

```
proxy<const_object_attribute> attr(char const*) const;
proxy<object_attribute> attr(char const*);
```

要件

name が ntbs。

効果

`*this` の名前 *name* の属性にアクセスする。

戻り値

ターゲットに `object(*static_cast<U*>(this))` を、キーに *name* を束縛した `proxy` オブジェクト。

```
proxy<const_object_objattribute> attr(object const&) const;
proxy<object_objattribute> attr(object const&);
```

要件

name は文字列を保持する `object`。

効果

`*this` の名前 *name* の属性にアクセスする。

戻り値

ターゲットに `object(*static_cast<U*>(this))` を、キーに *name* を束縛した `proxy` オブジェクト。

```
template <class T>
proxy<const_object_item> operator[](T const& key) const;
template <class T>
proxy<object_item> operator[](T const& key);
```

効果

`*this` の *key* が示す要素にアクセスする。

戻り値

ターゲットに `object(*static_cast<U*>(this))` を、キーに `object(key)` を束縛した `proxy` オブジェクト。

```
template <class T, class V>
proxy<const_object_slice> slice(T const& start, V const& end) const
template <class T, class V>
proxy<object_slice> slice(T const& start, V const& end);
```

効果

`*this` の `std::make_pair(object(start), object(finish))` が示すスライスにアクセスする。

戻り値

ターゲットに `object(*static_cast<U*>(this))` を、キーに `std::make_pair(object(start), object(finish))` を束縛した `proxy` オブジェクト。

object クラス

目的は `object` が可能な限り Python の変数のように振舞うことである。これにより Python で動作する式は概して C++でも同じ方法で動作するはずである。`object` の大部分のインターフェイスは、基底クラス `object_operators<object>` とこのヘッダが定義

する自由関数が提供する。

object クラスの概要

```
namespace boost { namespace python { namespace api
{
  class object : public object_operators<object>
  {
   public:
      object();

      object(object const&);

      template <class T>
      explicit object(T const& x);

      ~object();

      object& operator=(object const&);

      PyObject* ptr() const;

      bool is_none() const;
  };
}}}
```

object クラスのコンストラクタおよびデストラクタ

```
object();
```

効果

Python の `None` オブジェクトへの参照を管理するオブジェクトを構築する。

例外

なし。

```
template <class T>
explicit object(T const& x);
```

効果

x を Python に変換し、それへの参照を管理する。

例外

上記の変換が不可能な場合、`error_already_set`（Python の `TypeError` 例外を設定する）。

```
~object();
```

効果

内部で保持するオブジェクトの参照カウントを減らす。

object クラスの変更メソッド

```
object& operator=(object const& rhs);
```

効果

rhs が保持するオブジェクトの参照カウントを増やし、`*this` が保持するオブジェクトの参照カウントを減らす。

object クラスのオブザーバ

```
PyObject* ptr() const;
```

戻り値

内部で保持している Python オブジェクトへのポインタ。

```
bool is_none() const;
```

戻り値

`(ptr() == Py_None)` の結果。

proxy クラステンプレート

`object` に対する属性、要素およびスライスアクセスを実装するために、このドキュメントで述べた種々のポリシー（Policies）とともにこのテンプレートをインスタンス化する。`Policies::key_type` 型のオブジェクトを格納する。

proxy クラステンプレートの概要

```
namespace boost { namespace python { namespace api
  template <class Policies>
  class proxy : public object_operators<proxy<Policies> >
  {
   public:
      operator object() const;

      proxy const& operator=(proxy const&) const;
      template <class T>
      inline proxy const& operator=(T const& rhs) const;

      void del() const;

      template <class R>
      proxy operator+=(R const& rhs);
      template <class R>
      proxy operator-=(R const& rhs);
      template <class R>
      proxy operator*=(R const& rhs);
      template <class R>
      proxy operator/=(R const& rhs);
```

```
      template <class R>
      proxy operator%=(R const& rhs);
      template <class R>
      proxy operator<<=(R const& rhs);
      template <class R>
      proxy operator>>=(R const& rhs);
      template <class R>
      proxy operator&=(R const& rhs);
      template <class R>
      proxy operator|=(R const& rhs);
  };
}}}
```

proxy クラステンプレートのオブザーバ関数

```
operator object() const;
```

効果

`Policies::get(target, key)`にプロキシのターゲットオブジェクトとキーオブジェクトを適用する。

proxy クラステンプレートの変更関数

```
proxy const& operator=(proxy const&) const;
template <class T>
inline proxy const& operator=(T const& rhs) const;
```

効果

プロキシのターゲットオブジェクトとキーオブジェクトを使用して `Policies::set(target, key, object(rhs))`。

```
template <class R>
proxy operator+=(R const& rhs);
template <class R>
proxy operator-=(R const& rhs);
template <class R>
proxy operator*=(R const& rhs);
template <class R>
proxy operator/=(R const& rhs);
template <class R>
proxy operator%=(R const& rhs);
template <class R>
proxy operator<<=(R const& rhs);
template <class R>
proxy operator>>=(R const& rhs);
template <class R>
proxy operator&=(R const& rhs);
template <class R>
proxy operator|=(R const& rhs);
```

効果

与えられた operator@=について、`object(*this) @= rhs;`

戻り値

*this

```
void del() const;
```

効果

プロキシのターゲットオブジェクトとキーオブジェクトを使用して Policies::del(target, key)。

関数

```
template <class T>
void del(proxy<T> const& x);
```

効果

x.del()

```
template<class L,class R> object operator>(L const&l,R const&r);
template<class L,class R> object operator>=(L const&l,R const&r);
template<class L,class R> object operator<(L const&l,R const&r);
template<class L,class R> object operator<=(L const&l,R const&r);
template<class L,class R> object operator==(L const&l,R const&r);
template<class L,class R> object operator!=(L const&l,R const&r);
```

効果

Python 内で演算子をそれぞれ object(l) および object(r) に適用した結果を返す。

```
template<class L,class R> object operator+(L const&l,R const&r);
template<class L,class R> object operator-(L const&l,R const&r);
template<class L,class R> object operator*(L const&l,R const&r);
template<class L,class R> object operator/(L const&l,R const&r);
template<class L,class R> object operator%(L const&l,R const&r);
template<class L,class R> object operator<<(L const&l,R const&r);
template<class L,class R> object operator>>(L const&l,R const&r);
template<class L,class R> object operator&(L const&l,R const&r);
template<class L,class R> object operator^(L const&l,R const&r);
template<class L,class R> object operator|(L const&l,R const&r);
```

効果

Python 内で演算子をそれぞれ object(l) および object(r) に適用した結果を返す。

```
template<class R> object& operator+=(object&l,R const&r);
template<class R> object& operator-=(object&l,R const&r);
template<class R> object& operator*=(object&l,R const&r);
template<class R> object& operator/=(object&l,R const&r);
template<class R> object& operator%=(object&l,R const&r);
template<class R> object& operator<<=(object&l,R const&r)
template<class R> object& operator>>=(object&l,R const&r);
template<class R> object& operator&=(object&l,R const&r);
template<class R> object& operator^=(object&l,R const&r);
```

```
template<class R> object& operator|=(object&l,R const&r);
```

効果

対応する Python の複合演算子をそれぞれ *l* および `object(r)` に適用した結果を *l* に代入する。

結果

l

```
inline long len(object const& obj);
```

効果

`PyObject_Length(obj.ptr())`

戻り値

オブジェクトの `len()`

例

Python のコード:

```
def sum_items(seq):
   result = 0
   for x in seq:
      result += x
   return result
```

C++版:

```
object sum_items(object seq)
{
   object result = object(0);
   for (int i = 0; i < len(seq); ++i)
      result += seq[i];
   return result;
}
```

ヘッダ<boost/python/str.hpp>

はじめに

Python の `str` 型に対する TypeWrapper をエクスポートする。

クラス

str クラス

Python の組み込み `str` 型の文字列メソッドをエクスポートする。文字の範囲から `str` オブジェクトを構築する 2 引数コンストラクタ

以外の以下に定義するコンストラクタとメンバ関数のセマンティクスを完全に理解するには、TypeWrapper コンセプトの定義を読むことである。str は object から公開派生しているので、object の公開インターフェイスは str のインスタンスにも当てはまる。

str クラスの概要

```
namespace boost { namespace python
{
  class str : public object
  {
   public:
      str(); // str を新規作成する

      str(char const* s); // str を新規作成する

      str(char const* start, char const* finish); // str を新規作成する
      str(char const* start, std::size_t length); // str を新規作成する

      template <class T>
      explicit str(T const& other);

      str capitalize() const;

      template <class T>
      str center(T const& width) const;

      template<class T>
      long count(T const& sub) const;
      template<class T1, class T2>
      long count(T1 const& sub,T2 const& start) const;
      template<class T1, class T2, class T3>
      long count(T1 const& sub,T2 const& start, T3 const& end) const;

      object decode() const;
      template<class T>
      object decode(T const& encoding) const;
      template<class T1, class T2>
      object decode(T1 const& encoding, T2 const& errors) const;

      object encode() const;
      template <class T>
      object encode(T const& encoding) const;
      template <class T1, class T2>
      object encode(T1 const& encoding, T2 const& errors) const;

      template <class T>
      bool endswith(T const& suffix) const;
      template <class T1, class T2>
      bool endswith(T1 const& suffix, T2 const& start) const;
      template <class T1, class T2, class T3>
      bool endswith(T1 const& suffix, T2 const& start, T3 const& end) const;

      str expandtabs() const;
      template <class T>
      str expandtabs(T const& tabsize) const;

      template <class T>
      long find(T const& sub) const;
```

```
    template <class T1, class T2>
    long find(T1 const& sub, T2 const& start) const;
    template <class T1, class T2, class T3>
    long find(T1 const& sub, T2 const& start, T3 const& end) const;

    template <class T>
    long index(T const& sub) const;
    template <class T1, class T2>
    long index(T1 const& sub, T2 const& start) const;
    template <class T1, class T2, class T3>
    long index(T1 const& sub, T2 const& start, T3 const& end) const;

    bool isalnum() const;
    bool isalpha() const;
    bool isdigit() const;
    bool islower() const;
    bool isspace() const;
    bool istitle() const;
    bool isupper() const;

    template <class T>
    str join(T const& sequence) const;

    template <class T>
    str ljust(T const& width) const;

    str lower() const;
    str lstrip() const;

    template <class T1, class T2>
    str replace(T1 const& old, T2 const& new_) const;
    template <class T1, class T2, class T3>
    str replace(T1 const& old, T2 const& new_, T3 const& maxsplit) const;

    template <class T>
    long rfind(T const& sub) const;
    template <class T1, class T2>
    long rfind(T1 const& sub, T2 const& start) const;
    template <class T1, class T2, class T3>
    long rfind(T1 const& sub, T2 const& start, T3 const& end) const;

    template <class T>
    long rindex(T const& sub) const;
    template <class T1, class T2>
    long rindex(T1 const& sub, T2 const& start) const;
    template <class T1, class T2, class T3>
    long rindex(T1 const& sub, T2 const& start, T3 const& end) const;

    template <class T>
    str rjust(T const& width) const;

    str rstrip() const;

    list split() const;
    template <class T>
    list split(T const& sep) const;
    template <class T1, class T2>
    list split(T1 const& sep, T2 const& maxsplit) const;

    list splitlines() const;
    template <class T>
```

```
      list splitlines(T const& keepends) const;

      template <class T>
      bool startswith(T const& prefix) const;
      template <class T1, class T2>
      bool startswidth(T1 const& prefix, T2 const& start) const;
      template <class T1, class T2, class T3>
      bool startswidth(T1 const& prefix, T2 const& start, T3 const& end) const;

      str strip() const;
      str swapcase() const;
      str title() const;

      template <class T>
      str translate(T const& table) const;
      template <class T1, class T2>
      str translate(T1 const& table, T2 const& deletechars) const;

      str upper() const;
  };
}}
```

例

```
using namespace boost::python;
str remove_angle_brackets(str x)
{
  return x.strip('<').strip('>');
}
```

ヘッダ<boost/python/tuple.hpp>

はじめに

Python の tuple 型に対する TypeWrapper をエクスポートする。

クラス

tuple クラス

Python の組み込み tuple 型の文字列メソッドをエクスポートする。以下に定義するコンストラクタとメンバ関数のセマンティクスを完全に理解するには、TypeWrapper コンセプトの定義を読むことである。tuple は object から公開派生しているので、object の公開インターフェイスは tuple のインスタンスにも当てはまる。

tuple クラスの概要

```
namespace boost { namespace python
```

```
{
   class tuple : public object
   {
      // tuple()は空のtupleを作成
      tuple();

      // tuple(sequence)はシーケンスの要素で初期化したtupleを作成
      template <class T>
      explicit tuple(T const& sequence)
  };
}}
```

関数

make_tuple

```
namespace boost { namespace python
{
  tuple make_tuple();

  template <class A0>
  tuple make_tuple(A0 const& a0);

  template <class A0, class A1>
  tuple make_tuple(A0 const& a0, A1 const& a1);
  ...
  template <class A0, class A1,...class An>
  tuple make_tuple(A0 const& a0, A1 const& a1,...An const& an);
}}
```

`object(a0),...object(a`*n*`)`を組み合わせて新しいタプルオブジェクトを構築する。

例

```
using namespace boost::python;
tuple head_and_tail(object sequence)
{
    return make_tuple(sequence[0],sequence[-1]);
}
```

ヘッダ<boost/python/slice.hpp>

はじめに

Python の `slice` 型に対する TypeWrapper をエクスポートする。

クラス

slice クラス

組み込みの slice 型をラップして拡張スライシングプロトコルをエクスポートする。以下に定義するコンストラクタとメンバ関数のセマンティクスを完全に理解するには、TypeWrapper コンセプトの定義を読むことである。slice は object から公開派生しているので、object の公開インターフェイスは slice のインスタンスにも当てはまる。

slice クラスの概要

```
namespace boost { namespace python
{
  class slice : public object
  {
   public:
      slice(); // 空の slice を作成する。[::]と等価

      template <typename Int1, typename Int2>
      slice(Int1 start, Int2 stop);

      template <typename Int1, typename Int2, typename Int3>
      slice(Int1 start, Int2 stop, Int3 step);

      // この slice を作成したときの引数にアクセスする。
      object start();
      object stop();
      object step();

      // slice::get_indices()の戻り値の型
      template <typename RandomAccessIterator>
      struct range
      {
          RandomAccessIterator start;
          RandomAccessIterator stop;
          int step;
      };

      template <typename RandomAccessIterator>
      range<RandomAccessIterator>
      get_indices(
          RandomAccessIterator const& begin,
          RandomAccessIterator const& end);
  };
}}
```

slice クラスのコンストラクタ

```
slice();
```

効果

既定の stop、start、step 値で slice を構築する。Python の式 base[::]で作成したスライスオブジェクトと等価。

例外

なし。

```
template <typename Int1, typename Int2>
slice(Int1 start, Int2 stop);
```

要件

start および *stop* は slice_nil 型、または object 型へ変換可能。

効果

既定の step 値と与えられた *start*、*stop* 値で新しい slice を構築する。Python の組み込み関数 slice(start,stop)、または Python の式 base[start:stop]で作成したスライスオブジェクトと等価。

例外

引数を object 型へ変換できない場合、error_already_set（Python の TypeError 例外を設定する）。

```
template <typename Int1, typename Int2, typename Int3>
slice(Int1 start, Int2 stop, Int3 step);
```

要件

start、*stop* および *step* は slice_nil、または object 型へ変換可能。

効果

start、*stop*、*step* 値で新しい slice を構築する。Python の組み込み関数 slice(start,stop,step)、または Python の式 base[start:stop:step]で作成したスライスオブジェクトと等価。

例外

引数を object 型へ変換できない場合、error_already_set（Python の TypeError 例外を設定する）。

slice クラスのオブザーバ関数

```
object slice::start() const;
object slice::stop() const;
object slice::step() const;
```

効果

なし。

例外

なし。

戻り値

スライスを作成したときに使用した引数。スライスを作成したときに引数を省略したか slice_nil を使用した場合、その引数は PyNone への参照でありデフォルトコンストラクトされた object と等値である。原則的にはスライスオブジェクトを作成するのに任意

の型を使用できるが、現実的には大抵は整数である。

```
template <typename RandomAccessIterator>
slice::range<RandomAccessIterator>
slice::get_indices(
    RandomAccessIterator const& begin,
    RandomAccessIterator const& end) const;
```

引数

半開区間を形成する STL 準拠のランダムアクセスイテレータの組。

効果

引数の `[begin,end)` 範囲内の完全閉範囲を定義する RandomAccessIterator の組を作成する。`PyNone` か負の添字の効果、および 1 以外の step サイズをどう扱うか説明を求められたときに、この関数がスライスの添字を変換する。

戻り値

非 0 の step 値と、この関数の引数が与える範囲を指し閉区間を定義する RandomAccessIterator の組で初期化した `slice::range`。

例外

このスライスの引数が `PyNone` への参照でも `int` へ変換可能でもない場合、Python の `TypeError` 例外を送出する。結果の範囲が空の場合、`std::invalid_argument` を投げる。`slice::get_indices()` を呼び出すときは常に `try { ...; } catch (std::invalid_argument) {}` で囲んでこのケースを処理し適切に処置しなければならない。

根拠

閉区間。開空間を使ったとすると、step サイズが 1 以外の場合、終端のイテレータが末尾の直後より後方の位置や指定した範囲より前方を指す状態が必要となる。

空のスライスに対する例外。空の範囲を表す閉区間を定義することは不可能であるので、未定義の動作を防止するために他の形式のエラーチェックが必要となる。例外が捕捉されない場合、単に既定の例外処理機構により Python に変換される。

例

```
using namespace boost::python;

// Python リストの拡張スライス。
// 警告：組み込み型に対する拡張スライシングは Python 2.3 より前ではサポートされていない
list odd_elements(list l)
{
    return l[slice(_,_,2)];
}

// Numeric.array の多次元拡張スライスをとる
numeric::array even_columns(numeric::array arr)
{
    // 2 次元配列の 2 番目から 1 列おきに選択。
    // Python の "return arr[:, 1::2]" と等価。
    return arr[make_tuple( slice(), slice(1,_,2))];
}
```

```
// std::vector のスライスに対して合計をとる。
double partial_sum(std::vector<double> const& Foo, const slice index)
{
    slice::range<std::vector<double>::const_iterator> bounds;
    try {
        bounds = index.get_indices<>(Foo.begin(), Foo.end());
    }
    catch (std::invalid_argument) {
        return 0.0;
    }
    double sum = 0.0;
    while (bounds.start != bounds.stop) {
        sum += *bounds.start;
        std::advance( bounds.start, bounds.step);
    }
    sum += *bounds.start;
    return sum;
}
```

関数の呼び出しと作成

ヘッダ<boost/python/args.hpp>

はじめに

ラップしたC++関数にキーワード引数を指定する多重定義関数群を提供する。

keyword-expression

keyword-expression の結果は ntbs の列を保持するオブジェクトであり、その型は指定したキーワードの数を符号化する。*keyword-expression* は保持する一部またはすべてのキーワードについて既定値を持つことが可能である。

クラス

arg クラス

`arg` クラスのオブジェクトは1つのキーワードを保持する（サイズが1である）*keyword-expression* である。

arg クラスの概要

```
namespace boost { namespace python
{
        struct arg
        {
          template <class T>
                  arg &operator = (T const &value);
          explicit arg (char const *name){elements[0].name = name;}
```

```
        };

}}
```

arg クラスのコンストラクタ

```
arg(char const* name);
```

要件

引数は ntbs でなければならない。

効果

名前 *name* のキーワードを保持する `arg` オブジェクトを構築する。

arg クラスの operator=

```
template <class T> arg &operator = (T const &value);
```

要件

引数は Python へ変換可能でなければならない。

効果

キーワードの既定値を代入する。

戻り値

`this` への参照。

keyword-expression の operator ,

```
keyword-expression operator , (keyword-expression, const arg &kw) const
keyword-expression operator , (keyword-expression, const char *name) const;
```

要件

引数 *name* は ntbs でなければならない。

効果

1 つ以上のキーワードで *keyword-expression* 引数を拡張する。

戻り値

拡張した *keyword-expression*。

関数（非推奨）

args(...)

```
unspecified1 args(char const*);
unspecified2 args(char const*, char const*);
   .
   .
   .
unspecifiedN args(char const*, char const*, ... char const*);
```

要件

引数はすべて ntbs でなければならない。

戻り値

渡した引数をカプセル化する *keyword-expression* を表すオブジェクト。

例

```
#include <boost/python/def.hpp>
using namespace boost::python;

int f(double x, double y, double z=0.0, double w=1.0);

BOOST_PYTHON_MODULE(xxx)
{
   def("f", f
            , ( arg("x"), "y", arg("z")=0.0, arg("w")=1.0 )
            );
}
```

ヘッダ<boost/python/call.hpp>

はじめに

<boost/python/call.hpp>は、C++から Python の呼び出し可能オブジェクトを起動する call 関数テンプレート多重定義群を定義する。

関数

call

```
template <class R, class A1, class A2, ... class An>
R call(PyObject* callable, A1 const&, A2 const&, ... An const&)
```

要件

R はポインタ型、参照型、またはアクセス可能なコピーコンストラクタを持つ完全型。

効果

Python 内で `callable(a1, a2, ...an)` を起動する。`a1...an` は `call()` に対する引数で Python のオブジェクトに変換する。

戻り値

Python の呼び出し結果を C++の型 *R* に変換したもの。

根拠

完全なセマンティクスの説明と根拠については、このページを見よ。

例

以下の C++関数は、Python の呼び出し可能オブジェクトをその 2 つの引数に適用し結果を返す。Python の例外が送出した場合や結果を `double` に変換できない場合は例外を投げる。

```
double apply2(PyObject* func, double x, double y)
{
   return boost::python::call<double>(func, x, y);
}
```

ヘッダ<boost/python/call_method.hpp>

はじめに

`<boost/python/call_method.hpp>`は、C++から Python の呼び出し可能属性を起動する `call_method` 関数テンプレート多重定義群を定義する。

関数

call_method

```
template <class R, class A1, class A2, ... class An>
R call_method(PyObject* self, char const* method, A1 const&, A2 const&, ... An const&)
```

要件

R はポインタ型、参照型、またはアクセス可能なコピーコンストラクタを持つ完全型。

効果

Python 内で `self.method(a1, a2, ...an)` を起動する。`a1...an` は `call_method()` に対する引数で Python のオブジェクトに変換する。完全なセマンティクスの説明については、このページを見よ。

戻り値

Python の呼び出し結果を C++の型 *R* に変換したもの。

根拠

下の例で見るように、Python でオーバーライド可能な C++仮想関数を実装するのに重要である。

例

以下の C++コードは、`call_method` を使用して Python でオーバーライド可能な仮想関数を持つクラスをラップする方法を示している。

C++のモジュール定義

```
#include <boost/python/module.hpp>
#include <boost/python/class.hpp>
#include <boost/utility.hpp>
#include <cstring>

// ラップするクラス
class Base
{
 public:
   virtual char const* class_name() const { return "Base"; }
   virtual ~Base();
};

bool is_base(Base* b)
{
   return !std::strcmp(b->class_name(), "Base");
}

// ここからラッパコード
using namespace boost::python;

// コールバッククラス
class Base_callback : public Base
{
 public:
   Base_callback(PyObject* self) : m_self(self) {}

   char const* class_name() const { return call_method<char const*>(m_self, "class_name"); }
   char const* Base_name() const { return Base::class_name(); }
 private:
   PyObject* const m_self;
};

using namespace boost::python;
BOOST_PYTHON_MODULE(my_module)
{
    def("is_base", is_base);

    class_<Base,Base_callback, noncopyable>("Base")
        .def("class_name", &Base_callback::Base_name)
        ;

}
```

Python のコード

```
>>> from my_module import *
>>> class Derived(Base):
...     def __init__(self):
...         Base.__init__(self)
...     def class_name(self):
...         return self.__class__.__name__
...
>>> is_base(Base()) # C++から class_name()メソッドを呼び出す
1
>>> is_base(Derived())
0
```

ヘッダ<boost/python/data_members.hpp>

はじめに

make_getter()および make_setter()は、class_<>::def_readonly および class_<>::def_readwrite が C++データメンバをラップする Python の呼び出し可能オブジェクトを生成するために内部的に使用する関数である。

関数

make_getter 関数

```
template <class C, class D>
object make_getter(D C::*pm);

template <class C, class D, class Policies>
object make_getter(D C::*pm, Policies const& policies);
```

要件

Policies は CallPolicies のモデル。

効果

*C** へ from_python 変換可能な引数 1 つをとり、*C* オブジェクトの対応メンバ *D* を to_python 変換したものを返す Python の呼び出し可能オブジェクトを作成する。*policies* が与えられた場合、ここに述べるとおり関数に適用される。それ以外の場合、ライブラリは *D* がユーザ定義クラス型か判断し、そうであれば *Policies* に対して return_internal_reference<>を使用する。*D* がスマートポインタ型の場合、このテストで return_internal_reference<>が不適当に選択される可能性があることに注意していただきたい。これは既知の欠陥である。

戻り値

新しい Python の呼び出し可能オブジェクトを保持する object のインスタンス。

```
template <class D>
object make_getter(D const& d);
template <class D, class Policies>
object make_getter(D const& d, Policies const& policies);

template <class D>
object make_getter(D const* p);
template <class D, class Policies>
object make_getter(D const* p, Policies const& policies);
```

要件

Policies は CallPolicies のモデル。

効果

引数をとらず、必要に応じて `to_python` 変換した *d* か `*p` を返す Python の呼び出し可能オブジェクトを作成する。*policies* が与えられた場合、ここに述べるとおり関数に適用される。それ以外の場合、ライブラリは *D* がユーザ定義クラス型か判断し、そうであれば *Policies* に対して `reference_existing_object` を使用する。

戻り値

新しい Python の呼び出し可能オブジェクトを保持する `object` のインスタンス。

make_setter 関数

```
template <class C, class D>
object make_setter(D C::*pm);

template <class C, class D, class Policies>
object make_setter(D C::*pm, Policies const& policies);
```

要件：*Policies* は CallPolicies のモデル。

効果：Python の呼び出し可能オブジェクトを作成する。このオブジェクトは Python から呼び出されるときに `C*` と `D const&` にそれぞれ `from_python` 変換可能な 2 つの引数をとり、*C* オブジェクトの対応する *D* メンバを設定する。*policies* が与えられた場合、ここに述べるとおり関数に適用される。

戻り値：新しい Python の呼び出し可能オブジェクトを保持する `object` のインスタンス。

```
template <class D>
object make_setter(D& d);
template <class D, class Policies>
object make_setter(D& d, Policies const& policies);

template <class D>
object make_setter(D* p);
template <class D, class Policies>
object make_setter(D* p, Policies const& policies);
```

要件

Policies は CallPolicies のモデル。

効果

Python から `D const&` に変換され、*d* または `*p` に書き込まれる 1 つの引数を受け取る Python の呼び出し可能オブジェクトを作成する。*policies* が与えられた場合、ここに述べるとおり関数に適用される。

戻り値

新しい Python の呼び出し可能オブジェクトを保持する `object` のインスタンス。

例

以下のコードは、`make_getter` および `make_setter` を使用してデータメンバを関数としてエクスポートする。

```
#include <boost/python/data_members.hpp>
#include <boost/python/module.hpp>
#include <boost/python/class.hpp>

struct X
{
    X(int x) : y(x) {}
    int y;
};

using namespace boost::python;

BOOST_PYTHON_MODULE_INIT(data_members_example)
{
    class_<X>("X", init<int>())
        .def("get", make_getter(&X::y))
        .def("set", make_setter(&X::y))
        ;
}
```

Python から次のように使用する。

```
>>> from data_members_example import *
>>> x = X(1)
>>> x.get()
1
>>> x.set(2)
>>> x.get()
2
```

ヘッダ<boost/python/make_function.hpp>

はじめに

`make_function()` および `make_constructor()` は、`def()` および `class_<>::def()` が C++の関数およびメンバ関数をラップする Python の呼び出し可能オブジェクトを生成するのに内部的に使用する関数である。

関数

make_function

```
template <class F>
object make_function(F f)

template <class F, class Policies>
object make_function(F f, Policies const& policies)

template <class F, class Policies, class KeywordsOrSignature>
object make_function(F f, Policies const& policies, KeywordsOrSignature const& ks)

template <class F, class Policies, class Keywords, class Signature>
object make_function(F f, Policies const& policies, Keywords const& kw, Signature const& sig)
```

要件

F は関数ポインタ、またはメンバ関数ポインタ型。*policies* が与えられた場合、CallPolicies のモデルでなければならない。*keywords* が与えられた場合、*f* の引数長を超えない *keyword-expression* の結果でなければならない。

効果

Python の呼び出し可能オブジェクトを作成する。このオブジェクトは Python から呼び出されると、引数を C++に変換して *f* を呼び出す。*F* がメンバ関数へのポインタ型の場合、Python の第 1 引数が関数呼び出しの対象オブジェクト（`*this`）となり、残りの Python 引数は *f* に対する引数となる。

- *policies* が与えられた場合、ここに述べるとおり関数に適用する。
- *keywords* が与えられた場合、結果の関数における最後の引数に適用する。
- *Signature* が与えられた場合、MPL の先頭拡張可能列のインスタンスでなければならない。列の先頭が関数の戻り値型、後続が引数の型である。シグニチャが推論できない関数オブジェクト型をラップする場合や、ラップする関数に渡す型をオーバーライドしたい場合は *Signature* を渡すとよい。

戻り値

新しい Python の呼び出し可能オブジェクトを保持する `object` のインスタンス。

注意

ポインタ型の引数は、Python から `None` が渡された場合に `0` となる可能性がある。`const` な参照型の引数は、ラップした関数を呼び出す間だけに生存する Python オブジェクトから作成された一時オブジェクトを指す可能性がある。例えば Python のリストからの変換過程で生成した `std::vector` がそうである。永続的な lvalue が必要な場合は、非 `const` 参照の引数を使うとよい。

make_constructor

```
template <class F>
object make_constructor(F f)
```

```
template <class F, class Policies>
object make_constructor(F f, Policies const& policies)

template <class F, class Policies, class KeywordsOrSignature>
object make_constructor(F f, Policies const& policies, KeywordsOrSignature const& ks)

template <class F, class Policies, class Keywords, class Signature>
object make_constructor(F f, Policies const& policies, Keywords const& kw, Signature const& sig)
```

要件

F は関数ポインタ型。*policies* が与えられた場合、CallPolicies のモデルでなければならない。*keywords* が与えられた場合、*f* の引数長を超えない *keyword-expression* の結果でなければならない。

効果

Python から呼び出されると引数を C++に変換して *f* を呼び出す、Python の呼び出し可能オブジェクトを作成する。

戻り値

新しい Python の呼び出し可能オブジェクトを保持する object のインスタンス。

例

以下でエクスポートする C++関数は、2 つの関数のうち 1 つをラップする呼び出し可能オブジェクトを返す。

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

char const* foo() { return "foo"; }
char const* bar() { return "bar"; }

using namespace boost::python;
object choose_function(bool selector)
{
    if (selector)
        return boost::python::make_function(foo);
    else
        return boost::python::make_function(bar);
}

BOOST_PYTHON_MODULE(make_function_test)
{
    def("choose_function", choose_function);
}
```

Python からは次のように使用する。

```
>>> from make_function_test import *
>>> f = choose_function(1)
>>> g = choose_function(0)
>>> f()
'foo'
>>> g()
'bar'
```

ヘッダ<boost/python/overloads.hpp>

はじめに

C++の関数、メンバ関数および多重定義群から既定の引数とともに Python の関数および拡張クラスのメソッドの多重定義群を生成する機能を定義する。

overload-dispatch-expression

overload-dispatch-expression は、拡張クラスのために生成するメソッドの多重定義群を記述するのに使用する。以下のプロパティを持つ。

docstring	メソッドの__doc__属性に束縛される値を持つ ntbs
keywords	生成するメソッドの引数（の最後列）に名前を付ける *keyword-expression*。
call policies	CallPolicies モデルの何らかの型。
minimum arity	生成するメソッド多重定義が受け付ける引数の最小数。
maximum arity	生成するメソッド多重定義が受け付ける引数の最大数。

OverloadDispatcher コンセプト

OverloadDispatcher の x とは、minimum arity と maximum arity を持つクラスであり、以下がそれぞれ OverloadDispatcher と同じ最大・最小引数長を持つ合法な *overload-dispatch-expression* である。

```
X()
X(docstring)
X(docstring, keywords)
X(keywords, docstring)
X()[policies]
X(docstring)[policies]
X(docstring, keywords)[policies]
X(keywords, docstring)[policies]
```

- *policies* が与えられた場合、CallPolicies モデルの型でなければならず、結果の呼び出しポリシーとなる。それ以外の場合、結果の呼び出しポリシーは `default_call_policies` のインスタンスである。
- *docstring* が与えられた場合、ntbs でなければならず、結果のドキュメンテーション文字列となる。それ以外の場合、結果のドキュメンテーション文字列は空である。
- *keywords* が与えられた場合、長さが x の最大引数長以下である *keyword-expression* の結果でなければならず、結果のキーワード引数列となる。それ以外の場合、結果のキーワード引数列は空である。

マクロ

BOOST_PYTHON_FUNCTION_OVERLOADS(name, func_id, min_args, max_args)

現在のスコープで名前 *name* の OverloadDispatcher の定義へ展開する。これは以下の関数呼び出し生成に使用できる（`min_args` <= *i* <= `max_args`）。

```
func_id(a1, a2,...ai);
```

BOOST_PYTHON_MEMBER_FUNCTION_OVERLOADS(name, member_name, min_args, max_args)

現在のスコープで名前 *name* の OverloadDispatcher の定義へ展開する。これは以下の関数呼び出し生成に使用できる（`min_args` <= *i* <= `max_args`、`x` はクラス型オブジェクトへの参照）。

```
x.member_name(a1, a2,...ai);
```

例

```
#include <boost/python/module.hpp>
#include <boost/python/def.hpp>
#include <boost/python/args.hpp>
#include <boost/python/tuple.hpp>
#include <boost/python/class.hpp>
#include <boost/python/overloads.hpp>
#include <boost/python/return_internal_reference.hpp>

using namespace boost::python;

tuple f(int x = 1, double y = 4.25, char const* z = "wow")
{
    return make_tuple(x, y, z);
}

BOOST_PYTHON_FUNCTION_OVERLOADS(f_overloads, f, 0, 3)

struct Y {};
struct X
{
    Y& f(int x, double y = 4.25, char const* z = "wow")
    {
        return inner;
    }
    Y inner;
};

BOOST_PYTHON_MEMBER_FUNCTION_OVERLOADS(f_member_overloads, f, 1, 3)

BOOST_PYTHON_MODULE(args_ext)
{
    def("f", f,
        f_overloads(
```

```
            args("x", "y", "z"), "これはfのドキュメンテーション文字列"
        ));


    class_<Y>("Y")
        ;

    class_<X>("X", "これはXのドキュメンテーション文字列")
        .def("f1", &X::f,
                f_member_overloads(
                    args("x", "y", "z"), "fのドキュメンテーション文字列"
                )[return_internal_reference<>()]
        )
        ;
}
```

ヘッダ<boost/python/ptr.hpp>

はじめに

<boost/python/ptr.hpp>はptr()関数テンプレートを定義する。これによりオーバーライド可能な仮想関数の実装、Pythonの呼び出し可能オブジェクトの起動、あるいはC++オブジェクトからPythonへの明示的な変換においてC++のポインタ値をPythonに変換する方法を指定できる。通常、Pythonのコールバックにポインタを渡すと、Pythonのオブジェクトが懸垂参照を保持しないようポインタ先はコピーされる。新しいPythonのオブジェクトが単にポインタpのコピーを持つべきだということを指定するには、ユーザはpを直接渡すのではなくptr(p)を渡すようにする。このインターフェイスは同様に参照のコピーを抑止するboost::ref()の類似品である。

ptr(p)はis_pointer_wrapper<>メタ関数で検出可能なpointer_wrapper<>のインスタンスを返す。unwrap_pointer<>はpointer_wrapper<>から元のポインタ型を抽出するメタ関数である。これらのクラスは実装の詳細と考えてよい。

関数

ptr

```
template <class T>
pointer_wrapper<T> ptr(T x);
```

要件

*T*がポインタ型。

戻り値

pointer_wrapper<T>(x)

例外

なし。

クラス

pointer_wrapper クラステンプレート

ptr()が返す「型の封筒（envelope）」であり、Python のコールバックへ渡すポインタの参照セマンティクスを示すのに使用する。

pointer_wrapper クラステンプレートの概要

```
namespace boost { namespace python
{
    template<class Ptr> class pointer_wrapper
    {
     public:
        typedef Ptr type;

        explicit pointer_wrapper(Ptr x);
        operator Ptr() const;
        Ptr get() const;
    };
}}
```

pointer_wrapper クラステンプレートの型

```
typedef Ptr type;
```

ラップするポインタの型。

pointer_wrapper クラステンプレートのコンストラクタおよびデストラクタ

```
explicit pointer_wrapper(Ptr x);
```

要件

Ptr がポインタ型。

効果

pointer_wrapper<>に *x* を格納する。

例外

なし。

pointer_wrapper クラステンプレートのオブザーバ関数

```
operator Ptr() const;
Ptr get() const;
```

戻り値

格納しているポインタのコピー。

根拠

pointer_wrapper<>は実際のポインタ型の代理を意図しているが、時にはポインタを取得する明示的な方法があったほうがよい。

メタ関数

is_pointer_wrapper クラステンプレート

引数が pointer_wrapper<>である場合に value が真となる単項メタ関数。

is_pointer_wrapper クラステンプレートの概要

```
namespace boost { namespace python
{
    template<class T> class is_pointer_wrapper
    {
        static unspecified value = ...;
    };
}}
```

戻り値

T が pointer_wrapper<>の特殊化であれば true。value は未規定型の論理値へ変換可能な整数定数。

unwrap_pointer クラステンプレート

pointer_wrapper<>の特殊化からラップしたポインタ型を抽出する単項メタ関数。

unwrap_pointer クラステンプレートの概要

```
namespace boost { namespace python
{
    template<class T> class unwrap_pointer
    {
        typedef unspecified type;
    };
}}
```

戻り値

T が pointer_wrapper<>の特殊化の場合、T::type。それ以外の場合は T。

例

ptr()を使用してオブジェクトのコピーを抑止する例。

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

class expensive_to_copy
{
   ...
};

void pass_as_arg(expensive_to_copy* x, PyObject* f)
{
   // Python の関数 f を呼び出し、*x を「ポインタで」参照する
   // Python オブジェクトを渡す。
   //
   // *** 注意：*x を f() の引数として使用した後も延命させるのは ***
   // *** ユーザの責任である！失敗するとクラッシュする！       ***

   boost::python::call<void>(f, ptr(x));
}
...
```

ヘッダ<boost/python/raw_function.hpp>

はじめに

raw_function(...)は、tuple および dict を引数にとる関数を可変長の引数と任意のキーワード引数を受け取る Python の呼び出し可能オブジェクトに変換するのに使用する。

関数

raw_function

```
template <class F>
object raw_function(F f, std::size_t min_args = 0);
```

要件

f(tuple(), dict())が合法な形式。

戻り値

少なくとも *min_args* 個の引数を要求する呼び出し可能オブジェクト。呼び出されると実際の非キーワード引数列が tuple の第 1 引数として、キーワード引数列が dict の第 2 引数として *f* に渡される。

例

C++:

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

```
#include <boost/python/tuple.hpp>
#include <boost/python/dict.hpp>
#include <boost/python/module.hpp>
#include <boost/python/raw_function.hpp>
using namespace boost::python;

tuple raw(tuple args, dict kw)
{
    return make_tuple(args, kw);
}

BOOST_PYTHON_MODULE(raw_test)
{
    def("raw", raw_function(raw));
}
```

Python:

```
>>> from raw_test import *

>>> raw(3, 4, foo = 'bar', baz = 42)
((3, 4), {'foo': 'bar', 'baz': 42})
```

関数のドキュメンテーション

ヘッダ<boost/python/object/function_doc_signature.hpp>

はじめに

Boost.Python は、Python と C++シグニチャの自動的な連結を使ったドキュメンテーション文字列をサポートする。この機能は `class function_doc_signature_generator` が実装する。このクラスはユーザ定義のドキュメンテーション文字列の他に多重定義、与えられた引数の名前および既定値のすべてを使用して、与えられた関数のドキュメンテーション文字列を生成する。

クラス

function_doc_signature_generator クラス

このクラスは、1 つの関数の多重定義群に対するドキュメンテーション文字列のリストを返す公開関数を 1 つだけ持つ。

function_doc_signature_generator クラスの概要

```
namespace boost { namespace python { namespace objects {

    class function_doc_signature_generator
    {
      public:
          static list function_doc_signatures(function const *f);
```

```
    };

}}}
```

例

function_doc_signature_generator で生成したドキュメンテーション文字列

```
#include <boost/python/module.hpp>
#include <boost/python/def.hpp>
#include <boost/python/args.hpp>
#include <boost/python/tuple.hpp>
#include <boost/python/class.hpp>
#include <boost/python/overloads.hpp>
#include <boost/python/raw_function.hpp>

using namespace boost::python;

tuple f(int x = 1, double y = 4.25, char const* z = "wow")
{
    return make_tuple(x, y, z);
}

BOOST_PYTHON_FUNCTION_OVERLOADS(f_overloads, f, 0, 3)


struct X
{
    tuple f(int x = 1, double y = 4.25, char const* z = "wow")
    {
        return make_tuple(x, y, z);
    }
};

BOOST_PYTHON_MEMBER_FUNCTION_OVERLOADS(X_f_overloads, X::f, 0, 3)

tuple raw_func(tuple args, dict kw)
{
    return make_tuple(args, kw);
}

BOOST_PYTHON_MODULE(args_ext)
{
    def("f", f, (arg("x")=1, arg("y")=4.25, arg("z")="wow")
        , "これはfのドキュメンテーション文字列"
        );

    def("raw", raw_function(raw_func));

    def("f1", f, f_overloads("f1のドキュメンテーション文字列", args("x", "y", "z")));


    class_<X>("X", "これはXのドキュメンテーション文字列", init<>(args("self")))
        .def("f", &X::f
             , "これはX.fのドキュメンテーション文字列"
```

```
            , args("self","x", "y", "z"))

        ;

}
```

Python のコード:

```
>>> import args_ext
>>> help(args_ext)
Help on module args_ext:

NAME
    args_ext

FILE
    args_ext.pyd

CLASSES
    Boost.Python.instance(__builtin__.object)
        X

    class X(Boost.Python.instance)
     |  これはXのドキュメンテーション文字列
     |
     |  Method resolution order:
     |      X
     |      Boost.Python.instance
     |      __builtin__.object
     |
     |  Methods defined here:
     |
     |  __init__(...)
     |      __init__( (object)self) -> None :
     |       C++ signature:
     |           void __init__(struct _object *)
     |
     |  f(...)
     |      f( (X)self, (int)x, (float)y, (str)z) -> tuple : これはX.fのドキュメンテーション文字列
     |      C++ signature:
     |          class boost::python::tuple f(struct X {lvalue},int,double,char const *)
     |
     |    ................
     |
FUNCTIONS
    f(...)
        f([ (int)x=1 [, (float)y=4.25 [, (str)z='wow']]]) -> tuple : これはfのドキュメンテーション文字列
        C++ signature:
            class boost::python::tuple f([ int=1 [,double=4.25 [,char const *='wow']]])

    f1(...)
        f1([ (int)x [, (float)y [, (str)z]]]) -> tuple : f1のドキュメンテーション文字列
        C++ signature:
            class boost::python::tuple f1([ int [,double [,char const *]]])

    raw(...)
        object raw(tuple args, dict kwds) :
        C++ signature:
```

```
            object raw(tuple args, dict kwds)
```

ヘッダ<boost/python/converter/pytype_function.hpp>

はじめに

Python のシグニチャをサポートするには、変換器は関連する PyTypeObject へのポインタを返す get_pytype 関数を提供しなければならない。例として ResultConverter か to_python_converter を見るとよい。このヘッダファイルのクラスは get_pytype の実装に使用することを想定している。修飾無しの引数型とともに使用する class T に対するテンプレートの_direct 版[15]もある（これらはモジュールを読み込んだときに変換レジストリ内にあると考えてしかるべきものである）。

クラス

wrap_pytype クラス

このテンプレートは、テンプレート引数を返す静的メンバ get_pytype を生成する。

wrap_pytype クラスの概要

```
namespace boost { namespace python { namespace converter{

    template < PyTypeObject const *pytype >
    class wrap_pytype
    {
      public:
          static PyTypeObject const *get_pytype(){return pytype; }
    };

}}}
```

registered_pytype クラス

このテンプレートは、class_ で Python へエクスポートする型（修飾子があってもよい）のテンプレート引数とともに使用すべきである。生成された静的メンバ get_pytype は対応する Python の型を返す。

registered_pytype クラスの概要

```
namespace boost { namespace python { namespace converter{

    template < class T >
    class registered_pytype
    {
```

15 訳注 expected_from_python_type_direct、registered_pytype_direct、to_python_target_type_direct の 3 つ。

```
      public:
          static PyTypeObject const *get_pytype();
    };

}}}
```

expected_from_python_type クラス

このテンプレートは、型 T について登録済みの from_python 変換器を問い合わせし合致した Python 型を返す静的メンバ get_pytype を生成する。

expected_from_python_type クラスの概要

```
namespace boost { namespace python { namespace converter{

    template < class T >
    class expected_from_python_type
    {
      public:
          static PyTypeObject const *get_pytype();
    };

}}}
```

to_python_target_type クラス

このテンプレートは、T から変換可能な Python の型を返す静的メンバ get_pytype を生成する。

to_python_target_type クラスの概要

```
namespace boost { namespace python { namespace converter{

    template < class T >
    class to_python_target_type
    {
      public:
          static PyTypeObject const *get_pytype();
    };

}}}
```

例

以下の例では、Python のドキュメントにある標準的な noddy モジュール例を実装したとして、関連する宣言を noddy.h に置いたものと仮定する。noddy_NoddyObject は極限なまでに単純な拡張型であるので、この例は少しばかりわざとらしい。すべての情報がその戻り値の**型**に含まれる関数をラップしている。

C++のモジュール定義

```
#include <boost/python/reference.hpp>
#include <boost/python/module.hpp>
#include "noddy.h"

struct tag {};
tag make_tag() { return tag(); }

using namespace boost::python;

struct tag_to_noddy
#if defined BOOST_PYTHON_SUPPORTS_PY_SIGNATURES // Python のシグニチャがサポートされない場合は不要なオーバーヘッドが発生
: wrap_pytype<&noddy_NoddyType> // wrap_pytype から get_pytype を継承する
#endif
{
    static PyObject* convert(tag const& x)
    {
        return PyObject_New(noddy_NoddyObject, &noddy_NoddyType);
    }
};

BOOST_PYTHON_MODULE(to_python_converter)
{
    def("make_tag", make_tag);
    to_python_converter<tag, tag_to_noddy
#if defined BOOST_PYTHON_SUPPORTS_PY_SIGNATURES // Python のシグニチャがサポートされない場合は不正
          , true
#endif
          >(); // tag_to_noddy がメンバ get_pytype を持つので「真」
}
```

以下の例は、テンプレート `expected_from_python_type` および `to_python_target_type` を使用して Python との双方向変換器を登録している。

```
#include <boost/python/module.hpp>
#include <boost/python/def.hpp>
#include <boost/python/extract.hpp>
#include <boost/python/to_python_converter.hpp>
#include <boost/python/class.hpp>

using namespace boost::python;

struct A
{
};

struct B
{
  A a;
  B(const A& a_):a(a_){}
};

// A から Python の整数への変換器
struct BToPython
```

```
#if defined BOOST_PYTHON_SUPPORTS_PY_SIGNATURES // Python のシグニチャがサポートされていない場合は不要なオーバーヘッドが発生
   : converter::to_python_target_type<A>  // get_pytype を継承する
#endif
{
  static PyObject* convert(const B& b)
  {
    return incref(object(b.a).ptr());
  }
};

// Python の整数から A への変換
struct BFromPython
{
  BFromPython()
  {
    boost::python::converter::registry::push_back
        ( &convertible
        , &construct
        , type_id< B >()
#if defined BOOST_PYTHON_SUPPORTS_PY_SIGNATURES // Python のシグニチャがサポートされていない場合は不正
        , &converter::expected_from_python_type<A>::get_pytype// A へ変換可能なものは B へ変換可能
#endif
        );
  }

  static void* convertible(PyObject* obj_ptr)
  {
      extract<const A&> ex(obj_ptr);
      if (!ex.check()) return 0;
      return obj_ptr;
  }

  static void construct(
      PyObject* obj_ptr,
      converter::rvalue_from_python_stage1_data* data)
  {
    void* storage = (
        (converter::rvalue_from_python_storage< B >*)data)-> storage.bytes;

    extract<const A&> ex(obj_ptr);
    new (storage) B(ex());
    data->convertible = storage;
  }
};


B func(const B& b) { return b ; }

BOOST_PYTHON_MODULE(pytype_function_ext)
{
  to_python_converter< B , BtoPython
#if defined BOOST_PYTHON_SUPPORTS_PY_SIGNATURES // Python のシグニチャがサポートされていない場合は不正
              ,true
#endif
              >(); // get_pytype を持つ
  BFromPython();

  class_<A>("A") ;
```

```
  def("func", &func);

}



>>> from pytype_function_ext import *
>>> print func.__doc__
func( (A)arg1) -> A :
    C++ signature:
         struct B func(struct B)
```

CallPolicies のモデル

ヘッダ<boost/python/default_call_policies.hpp>

クラス

default_call_policies クラス

default_call_policies は precall および postcall の振る舞いを持たない CallPolicies のモデルであり、値返しを行う result_converter である。ラップする C++の関数およびメンバ関数は、特に指定しなければ default_call_policies を使用する。新規の CallPolicies は default_call_policies から派生すると便利である。

default_call_policies クラスの概要

```
namespace boost { namespace python
{
    struct default_call_policies
    {
        static bool precall(PyObject*);
        static PyObject* postcall(PyObject*, PyObject* result);
        typedef default_result_converter result_converter;
        template <class Sig> struct extract_return_type : mpl::front<Sig>{};
    };
}}
```

default_call_policies クラスの静的関数

```
bool precall(PyObject*);
```

戻り値

true

例外

なし

```
PyObject* postcall(PyObject*, PyObject* result);
```

戻り値

result

例外

なし

default_result_converter クラス

default_result_converter は、非ポインタ型、char const*または PyObject*を値で返す C++関数をラップするのに使用する ResultConverterGenerator モデルである。

default_result_converter クラスの概要

```
namespace boost { namespace python
{
    struct default_result_converter
    {
        template <class T> struct apply;
    };
}}
```

default_result_converter クラスのメタ関数

```
template <class T> struct apply
```

要件

*T*が参照型でない。*T*がポインタ型の場合、*T*は const char*か PyObject*。

戻り値

typedef to_python_value<T const&> type;

例

この例は Boost.Python の実装そのものからとった。return_value_policy クラステンプレートは precall および postcall に対する振る舞いの実装を持たないので、その基底クラスは default_call_policies となっている。

```
template <class Handler, class Base = default_call_policies>
struct return_value_policy : Base
{
   typedef Handler result_converter;
};
```

ヘッダ<boost/python/return_arg.hpp>

はじめに

`return_arg` および `return_self` のインスタンスは、ラップする（メンバ）関数の指定した引数（大抵は `*this`）を返す CallPolicies モデルである。

クラス

return_arg クラステンプレート

引数	要件	説明	既定
arg_pos	`std::size_t` 型の正のコンパイル時定数。	返す引数の位置。[16]	1
Base	CallPolicies のモデル	ポリシーの合成に使用。提供する `result_converter` は `return_arg` によりオーバーライドされるが、その `precall` および `postcall` ポリシーは CallPolicies の項に示すとおり合成される。	default_call_policies

return_arg クラステンプレートの概要

```
namespace boost { namespace python
{
   template <size_t arg_pos=1, class Base = default_call_policies>
   struct return_arg : Base
   {
      static PyObject* postcall(PyObject*, PyObject* result);
      struct result_converter{ template <class T> struct apply; };
      template <class Sig> struct extract_return_type : mpl::at_c<Sig, arg_pos>{};

   };
}}
```

return_arg クラスの静的関数

```
PyObject* postcall(PyObject* args, PyObject* result);
```

要件

`PyTuple_Check(args)` != 0 かつ `PyTuple_Size(args)` != 0

16 訳注 *arg_pos* テンプレート引数に 0 を指定することはできません。

戻り値

PyTuple_GetItem(args,arg_pos-1)

return_self クラステンプレート

return_self クラステンプレートの概要

```
namespace boost { namespace python
{
   template <class Base = default_call_policies>
   struct return_self
     : return_arg<1,Base>
   {};
}}
```

例

C++のモジュール定義

```
#include <boost/python/module.hpp>
#include <boost/python/class.hpp>
#include <boost/python/return_arg.hpp>

struct Widget
{
   Widget() :sensitive_(true){}
   bool get_sensitive() const { return sensitive_; }
   void set_sensitive(bool s) { this->sensitive_ = s; }
 private:
   bool sensitive_;
};

struct Label : Widget
{
   Label() {}

   std::string  get_label() const { return label_; }
   void set_label(const std::string &l){ label_ = l; }

 private:
   std::string label_;
};

using namespace boost::python;
BOOST_PYTHON_MODULE(return_self_ext)
{
   class_<widget>("Widget")
      .def("sensitive", &Widget::get_sensitive)
      .def("sensitive", &Widget::set_sensitive, return_self<>())
      ;

   class_<Label, bases<Widget> >("Label")
```

```
        .def("label", &Label::get_label)
        .def("label", &Label::set_label, return_self<>())
        ;
}
```

Python のコード

```
>>> from return_self_ext import *
>>> l1 = Label().label("foo").sensitive(false)
>>> l2 = Label().sensitive(false).label("foo")
```

ヘッダ<boost/python/return_internal_reference.hpp>

はじめに

`return_internal_reference` のインスタンスは、自由関数かメンバ関数の引数またはメンバ関数の対象が内部的に保持するオブジェクトへのポインタおよび参照を参照先のコピーを作成することなく安全に返すことを可能とする CallPolicies モデルである。第 1 テンプレート引数の既定値は、内包するオブジェクトがラップするメンバ関数の対象（`*this`）となるよくある場合を処理する。

クラス

return_internal_reference クラステンプレート

引数	要件	説明	既定
owner_arg	`std::size_t` 型の正のコンパイル時定数。	返す参照かポインタ先のオブジェクトを含む引数の添字。ラップするのがメンバ関数の場合、引数 1 は対象オブジェクト（`*this`）である。[17]対象の Python オブジェクト型が弱い参照をサポートしない場合、ラップする関数を呼び出すと Python の `TypeError` 例外を送出する。	1
Base	CallPolicies のモデルである	ポリシーの合成に使用。提供する `result_converter` は `return_internal_reference` によりオーバーライドされるが、その `precall` および `postcall` ポリシーは CallPolicies の項に示すとおり合成される。	`default_call_policies`

return_internal_reference クラステンプレートの概要

```
namespace boost { namespace python
{
```

17 訳注 *owner_arg* テンプレート引数に 0 や引数列の範囲を超える値を指定することはできません。

```
   template <std::size_t owner_arg = 1, class Base = default_call_policies>
   struct return_internal_reference : Base
   {
      static PyObject* postcall(PyObject*, PyObject* result);
      typedef reference_existing_object result_converter;
   };
}}
```

return_internal_reference クラスの静的関数

```
PyObject* postcall(PyObject* args, PyObject* result);
```

要件

PyTuple_Check(args) != 0

戻り値

with_custodian_and_ward_postcall::postcall(args, result)

例

C++のモジュール定義

```
#include <boost/python/module.hpp>
#include <boost/python/class.hpp>
#include <boost/python/return_internal_reference.hpp>

class Bar
{
 public:
   Bar(int x) : x(x) {}
   int get_x() const { return x; }
   void set_x(int x) { this->x = x; }
 private:
   int x;
};

class Foo
{
 public:
   Foo(int x) : b(x) {}

   // 内部的な参照を返す
   Bar const& get_bar() const { return b; }

 private:
   Bar b;
};

using namespace boost::python;
BOOST_PYTHON_MODULE(internal_refs)
{
   class_<Bar>("Bar", init<int>())
```

```
      .def("get_x", &Bar::get_x)
      .def("set_x", &Bar::set_x)
      ;

   class_<Foo>("Foo", init<int>())
      .def("get_bar", &Foo::get_bar
          , return_internal_reference<>())
      ;
}
```

Python のコード

```
>>> from internal_refs import *
>>> f = Foo(3)
>>> b1 = f.get_bar()
>>> b2 = f.get_bar()
>>> b1.get_x()
3
>>> b2.get_x()
3
>>> b1.set_x(42)
>>> b2.get_x()
42
```

ヘッダ<boost/python/return_value_policy.hpp>

はじめに

`return_value_policy` のインスタンスは、単純に ResultConverterGenerator と省略可能な `Base` CallPolicies を合成した CallPolicies モデルである。

クラス

return_value_policy クラステンプレート

引数	要件	既定
ResultConverterGenerator	ResultConverterGenerator のモデル	
Base	CallPolicies のモデル	`default_call_policies`

return_value_policy クラステンプレートの概要

```
namespace boost { namespace python
{
  template <class ResultConverterGenerator, class Base = default_call_policies>
  struct return_value_policy : Base
```

```
  {
      typedef ResultConverterGenerator result_converter;
  };
}}
```

例

C++のモジュール定義

```
#include <boost/python/module.hpp>
#include <boost/python/class.hpp>
#include <boost/python/copy_const_reference.hpp>
#include <boost/python/return_value_policy.hpp>

// ラップするクラス群
struct Bar { int x; }

struct Foo {
   Foo(int x) : { b.x = x; }
   Bar const& get_bar() const { return b; }
 private:
   Bar b;
};

// ラッパコード
using namespace boost::python;
BOOST_PYTHON_MODULE(my_module)
{
   class_<Bar>("Bar");

   class_<Foo>("Foo", init<int>())
      .def("get_bar", &Foo::get_bar
          , return_value_policy<copy_const_reference>())
      ;
}
```

Python のコード

```
>>> from my_module import *
>>> f = Foo(3)          # Foo オブジェクトを作成
>>> b = f.get_bar()     # 内部的な Bar オブジェクトのコピーを作成
```

ヘッダ<boost/python/with_custodian_and_ward.hpp>

はじめに

このヘッダは、関数の Python 引数および戻り値オブジェクトの 2 つの間の寿命依存性を確立する機能を提供する。**非後見人**

（ward）オブジェクトは**管理人**（custodian）オブジェクトが弱い参照をサポートしている限り（Boost.Python の拡張クラスはすべて弱い参照をサポートする）管理人オブジェクトが破壊されるまで破壊されない。**管理人**オブジェクトが弱い参照をサポートしておらず None でもない場合、適切な例外が投げられる。2 つのクラステンプレート with_custodian_and_ward および with_custodian_and_ward_postcall の違いはそれらが効果を発揮する場所である。

不注意で懸垂ポインタを作成してしまう可能性を減らすため、既定では寿命の束縛は背後にある C++オブジェクトが呼び出される**前に**行う。しかしながら結果のオブジェクトは呼び出すまで無効であるので、呼び出し後に寿命の束縛を行う with_custodian_and_ward_postcall を提供している。また with_custodian_and_ward<>::precall の後だが背後にある C++オブジェクトが実際にポインタを格納するより前に C++例外が投げられた場合、管理人オブジェクトと非後見人オブジェクトの寿命は意図的にともに束縛されるため、ラップする関数のセマンティクスに応じて代わりに with_custodian_and_ward_postcall を選択するとよい。

関数呼び出し境界を超えて生のポインタの**所有権を譲渡する**関数をラップする場合、これは適したツールでは**ない**ことに注意していただきたい。必要があればよくある質問と回答を参照されたい。

クラス

with_custodian_and_ward クラステンプレート

引数	要件	説明[18]	既定
custodian	std::size_t 型の正のコンパイル時定数。	確立する寿命関係において依存される側を指すテンプレート引数の 1 から始まる添字。メンバ関数をラップする場合、引数 1 は対象オブジェクト（*this）である。対象の Python オブジェクト型が弱い参照をサポートしない場合、ラップする C++オブジェクトを呼び出すと Python の TypeError 例外を送出することに注意していただきたい。	
ward	std::size_t 型の正のコンパイル時定数。	確立する寿命関係において依存する側を指すテンプレート引数の 1 から始まる添字。メンバ関数をラップする場合、引数 1 は対象オブジェクト（*this）である。	
Base	CallPolicies のモデル	ポリシーの合成に使用する。	default_call_policies

with_custodian_and_ward クラステンプレートの概要

```
namespace boost { namespace python
{
   template <std::size_t custodian, std::size_t ward, class Base = default_call_policies>
   struct with_custodian_and_ward : Base
   {
      static bool precall(PyObject* args);
```

18 訳注 *custodian* および *ward* テンプレート引数に 0 や引数列の範囲を超える値を指定することはできません。また両者に同じ値を指定することもできません。

```
    };
}}
```

with_custodian_and_ward クラスの静的関数

```
bool precall(PyObject* args);
```

要件

PyTuple_Check(args) != 0

効果

ward で指定した引数の寿命を *custodian* で指定した引数の寿命に依存させる。

戻り値

失敗時は false(PyErr_Occurred() != 0)。それ以外は true。

with_custodian_and_ward_postcall クラス

引数	要件	説明[19]	既定
custodian	std::size_t 型の正のコンパイル時定数。	確立する寿命関係において依存される側を指すテンプレート引数の添字。0 は戻り値、1 は第 1 引数を表す。メンバ関数をラップする場合、1 は対象オブジェクト(*this)である。対象の Python オブジェクト型が弱い参照をサポートしない場合、ラップする C++オブジェクトを呼び出すと Python の TypeError 例外を送出することに注意していただきたい。	
ward	std::size_t 型の正のコンパイル時定数。	確立する寿命関係において依存する側を指すテンプレート引数の添字。0 は戻り値、1 は第 1 引数を表す。メンバ関数をラップする場合、引数 1 は対象オブジェクト(*this)である。	
Base	CallPolicies のモデル	ポリシーの合成に使用する。	default_call_policies

with_custodian_and_ward_postcall クラステンプレートの概要

```
namespace boost { namespace python
{
   template <std::size_t custodian, std::size_t ward, class Base = default_call_policies>
   struct with_custodian_and_ward_postcall : Base
   {
      static PyObject* postcall(PyObject* args, PyObject* result);
   };
}}
```

19 訳注 *custodian* および *ward* テンプレート引数に引数列の範囲を超える値を指定することはできません。また両者に同じ値を指定することもできません。

with_custodian_and_ward_postcall クラスの静的関数

```
PyObject* postcall(PyObject* args, PyObject* result);
```

要件

PyTuple_Check(args) != 0 かつ result != 0。

効果

ward で指定した引数の寿命を *custodian* で指定した引数の寿命に依存させる。

戻り値

失敗時は 0（PyErr_Occurred() != 0）。それ以外は true。

例

以下はライブラリの return_internal_reference の実装に with_custodian_and_ward_postcall を使用している例である。

```
template <std::size_t owner_arg = 1, class Base = default_call_policies>
struct return_internal_reference
    : with_custodian_and_ward_postcall<0, owner_arg, Base>
{
   typedef reference_existing_object result_converter;
};
```

ResultConverter のモデル

ヘッダ<boost/python/to_python_indirect.hpp>

はじめに

<boost/python/to_python_indirect.hpp>は、ラップした C++クラスインスタンスをポインタかスマートポインタで保持する新しい Python オブジェクトを構築する手段を提供する。

クラス

to_python_indirect クラステンプレート

to_python_indirect クラステンプレートは第 1 引数型のオブジェクトを拡張クラスのインスタンスとして Python に変換する。第 2 引数で与えた所有権ポリシーを使用する。

以下の表で *x* は型 *T* のオブジェクト、*h* は boost::python::objects::instance_holder*型のオブジェクト、*p* は型 U*のオブジェクトである。

引数	要件	説明
T	`U` *cv*`&`（*cv* は省略可能な CV 指定子）か、`*x` が `U const&` に変換可能な Dereferenceable 型のいずれか（`U` はクラス型）。	`class_` クラステンプレートで Python へエクスポートする C++クラスを参照剥がしする型。
MakeHolder	`h = MakeHolder::execute(p);`	静的関数 `execute()` が `instance_holder` を作成するクラス。

`to_python_indirect` のインスタンスは ResultConverter のモデルである。

to_python_indirect synopsis クラステンプレート

```
namespace boost { namespace python
{
  template <class T, class MakeHolder>
  struct to_python_indirect
  {
     static bool convertible();
     PyObject* operator()(T ptr_or_reference) const;
   private:
     static PyTypeObject* type();
  };
}}
```

to_python_indirect クラステンプレートのオブザーバ関数

```
PyObject* operator()(T x) const;
```

要件

x はオブジェクトへの参照（ポインタ型の場合、非 null）。かつ `convertible() == true`。

効果

適切に型付けされた Boost.Python 拡張クラスのインスタンスを作成し、`MakeHolder` を使用して *x* から `instance_holder` を作成する。次に新しい拡張クラスインスタンス内に `instance_holder` をインストールし、最後にそれへのポインタを返す。

to_python_indirect クラステンプレートの静的関数

```
bool convertible();
```

効果

いずれかのモジュールが `U` に対応する Python 型を登録していれば `true`。

例

`reference_existing_object` の機能をコンパイル時のエラーチェックを省いて模造した例。

```
struct make_reference_holder
{
   typedef boost::python::objects::instance_holder* result_type;
   template <class T>
   static result_type execute(T* p)
   {
      return new boost::python::objects::pointer_holder<T*, T>(p);
   }
};

struct reference_existing_object
{
   // ResultConverter を返すメタ関数
   template <class T>
   struct apply
   {
      typedef boost::python::to_python_indirect<T,make_reference_holder> type;
   };
};
```

ヘッダ <boost/python/to_python_value.hpp>

クラス

to_python_value クラステンプレート

to_python_value は、引数を新しい Python オブジェクトにコピーする ResultConverter モデルである。

to_python_value クラスの概要

```
namespace boost { namespace python
{
   template <class T>
   struct to_python_value
   {
      typedef typename add_reference<
         typename add_const<T>::type
      >::type argument_type;

      static bool convertible();
      PyObject* operator()(argument_type) const;
   };
}}
```

to_python_value クラスのオブザーバ関数

```
static bool convertible();
```

戻り値

`T` から Python へ値による変換が可能な変換器が登録されていれば `true`。

```
PyObject* operator()(argument_type x) const;
```

要件

`convertible() == true`

効果

x を Python に変換する。

戻り値

`T` の変換器が登録されていれば、その結果の Python オブジェクト。それ以外の場合は `0`。

ResultConverterGenerator のモデル

ヘッダ<boost/python/copy_const_reference.hpp>

クラス

copy_const_reference クラス

`copy_const_reference` は、参照先の値を新しい Python オブジェクトにコピーする型への const 参照を返す C++関数をラップするのに使用する ResultConverterGenerator のモデルである。

copy_const_reference クラスの概要

```
namespace boost { namespace python
{
    struct copy_const_reference
    {
        template <class T> struct apply;
    };
}}
```

copy_const_reference クラスのメタ関数

```
template <class T> struct apply
```

要件

ある `U` に対して *T* が `U const&`。

戻り値

```
typedef to_python_value<T> type;
```

例

C++のモジュール定義

```
#include <boost/python/module.hpp>
#include <boost/python/class.hpp>
#include <boost/python/copy_const_reference.hpp>
#include <boost/python/return_value_policy.hpp>

// ラップするクラス群
struct Bar { int x; }

struct Foo {
   Foo(int x) : { b.x = x; }
   Bar const& get_bar() const { return b; }
 private:
   Bar b;
};

// ラッパコード
using namespace boost::python;
BOOST_PYTHON_MODULE(my_module)
{
    class_<Bar>("Bar");

     class_<Foo>("Foo", init<int>())
        .def("get_bar", &Foo::get_bar
            , return_value_policy<copy_const_reference>())
       ;
}
```

Python のコード

```
>>> from my_module import *
>>> f = Foo(3)          # Foo オブジェクトを作成
>>> b = f.get_bar()     # 内部的な Bar オブジェクトのコピーを作成
```

ヘッダ<boost/python/copy_non_const_reference.hpp>

クラス

copy_non_const_reference クラス

`copy_non_const_reference` は、参照先の値を新しい Python オブジェクトにコピーする型への非 const 参照を返す C++関数をラップするのに使用する ResultConverterGenerator のモデルである。

copy_non_const_reference クラスの概要

```
namespace boost { namespace python
{
    struct copy_non_const_reference
    {
        template <class T> struct apply;
    };
}}
```

copy_non_const_reference クラスのメタ関数

```
template <class T> struct apply
```

要件

非 const な U に対して *T* が U& である。

戻り値

```
typedef to_python_value<T> type;
```

例

C++のコード

```
#include <boost/python/module.hpp>
#include <boost/python/class.hpp>
#include <boost/python/copy_non_const_reference.hpp>
#include <boost/python/return_value_policy.hpp>

// ラップするクラス群
struct Bar { int x; }

struct Foo {
   Foo(int x) : { b.x = x; }
   Bar& get_bar() { return b; }
 private:
   Bar b;
};

// ラッパコード
using namespace boost::python;
BOOST_PYTHON_MODULE(my_module)
{
    class_<Bar>("Bar");

     class_<Foo>("Foo", init<int>())
        .def("get_bar", &Foo::get_bar
            , return_value_policy<copy_non_const_reference>())
       ;
}
```

Python のコード

```
>>> from my_module import *
>>> f = Foo(3)          # Foo オブジェクトを作成
>>> b = f.get_bar()     # 内部的な Bar オブジェクトのコピーを作成
```

ヘッダ<boost/python/manage_new_object.hpp>

クラス

manage_new_object クラス

manage_new_object は、new 式で確保したオブジェクトへのポインタを返し、呼び出し側がそのオブジェクトを削除する責任をもつことを想定する C++関数をラップするのに使用する ResultConverterGenerator のモデルである。

manage_new_object クラスの概要

```
namespace boost { namespace python
{
    struct manage_new_object
    {
        template <class T> struct apply;
    };
}}
```

manage_new_object クラスのメタ関数

```
template <class T> struct apply
```

要件

ある U に対して *T* が U*。

戻り値

typedef to_python_indirect<T> type;

例

C++側

```
#include <boost/python/module.hpp>
#include <boost/python/class.hpp>
#include <boost/python/manage_new_object.hpp>
```

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


struct Foo {
   Foo(int x) : x(x){}
   int get_x() { return x; }
   int x;
};

Foo* make_foo(int x) { return new Foo(x); }

// ラッパコード
using namespace boost::python;
BOOST_PYTHON_MODULE(my_module)
{
    def("make_foo", make_foo, return_value_policy<manage_new_object>())
    class_<Foo>("Foo")
        .def("get_x", &Foo::get_x)
        ;
}
```

Python 側

```
>>> from my_module import *
>>> f = make_foo(3)      # Foo オブジェクトを作成
>>> f.get_x()
3
```

ヘッダ<boost/python/reference_existing_object.hpp>

クラス

reference_existing_object クラス

reference_existing_object は、C++オブジェクトへの参照かポインタを返す C++関数をラップするのに使用する ResultConverterGenerator モデルである。ラップした関数を呼び出すとき、戻り値が参照する値はコピーされない。新しい Python オブジェクトは参照先へのポインタを持ち、対応する Python オブジェクトと少なくとも同じ長さの寿命となるような処置はなされない。よって、with_custodian_and_ward 等の CallPolicies モデルを利用した他の寿命管理無しで reference_existing_object を使用すると**非常に危険**となる可能性がある。このクラスは return_internal_reference の実装に使用されている。

reference_existing_object クラスの概要

```
namespace boost { namespace python
{
    struct reference_existing_object
    {
```

```
        template <class T> struct apply;
    };
}}
```

reference_existing_object クラスのメタ関数

```
template <class T> struct apply
```

要件

ある U に対して *T* が U&か U*。

戻り値

typedef to_python_indirect<T,V> type。V は、ラップする関数の戻り値が参照する先への所有権のない U*ポインタを持つインスタンスホルダを構築する execute 静的関数を持つクラス。

例

C++側

```
#include <boost/python/module.hpp>
#include <boost/python/class.hpp>
#include <boost/python/reference_existing_object.hpp>
#include <boost/python/return_value_policy.hpp>
#include <utility>

// ラップするクラス群
struct Singleton
{
   Singleton() : x(0) {}

   int exchange(int n)  // x を設定し、古い値を返す
   {
        std::swap(n, x);
        return n;
   }

   int x;
};

Singleton& get_it()
{
   static Singleton just_one;
   return just_one;
}

// ラッパコード
using namespace boost::python;
BOOST_PYTHON_MODULE(singleton)
{
    def("get_it", get_it,
```

```
        return_value_policy<reference_existing_object>());

    class_<Singleton>("Singleton")
       .def("exchange", &Singleton::exchange)
       ;
}
```

Python 側

```
>>> import singleton
>>> s1 = singleton.get_it()
>>> s2 = singleton.get_it()
>>> id(s1) == id(s2)  # s1 と s2 は同じオブジェクトではないが
0
>>> s1.exchange(42)   # 同じ C++の Singleton を参照する
0
>>> s2.exchange(99)
42
```

ヘッダ<boost/python/return_by_value.hpp>

クラス

return_by_value クラス

return_by_value は、戻り値が新しい Python オブジェクトへコピーされる参照型か値型を返す C++関数をラップするのに使用する ResultConverterGenerator モデルである。

return_by_value クラスの概要

```
namespace boost { namespace python
{
    struct return_by_value
    {
        template <class T> struct apply;
    };
}}
```

return_by_value クラスのメタ関数

```
template <class T> struct apply
```

戻り値

typedef to_python_value<T> type;

例

C++のモジュール定義

```
#include <boost/python/module.hpp>
#include <boost/python/class.hpp>
#include <boost/python/return_by_value.hpp>
#include <boost/python/return_value_policy.hpp>

// ラップするクラス群
struct Bar { };

Bar global_bar;

// ラップする関数群：
Bar b1();
Bar& b2();
Bar const& b3();

// ラッパコード
using namespace boost::python;
template <class R>
void def_void_function(char const* name, R (*f)())
{
   def(name, f, return_value_policy<return_by_value>());
}

BOOST_PYTHON_MODULE(my_module)
{
    class_<Bar>("Bar");
    def_void_function("b1", b1);
    def_void_function("b2", b2);
    def_void_function("b3", b3);
}
```

Python のコード

```
>>> from my_module import *
>>> b = b1() # これらの呼び出しは
>>> b = b2() # それぞれ新しい Bar オブジェクトを
>>> b = b3() # 個別に作成する
```

ヘッダ<boost/python/return_opaque_pointer.hpp>

クラス

return_opaque_pointer クラス

return_opaque_pointer は、新しい Python オブジェクトに戻り値がコピーされる未定義型へのポインタを返す C++関数をラップするのに使用する ResultConverterGenerator モデルである。

戻り値のポインタ先の型について type_id 関数の特殊化を定義するには、return_opaque_pointer ポリシーを指定することに加え、BOOST_PYTHON_OPAQUE_SPECIALIZED_TYPE_ID マクロを使用しなければならない。

return_opaque_pointer クラスの概要

```
namespace boost { namespace python
{
    struct return_opaque_pointer
    {
        template <class R> struct apply;
    };
}}
```

return_opaque_pointer クラスのメタ関数

```
template <class R> struct apply
```

戻り値

typedef detail::opaque_conversion_holder<R> type;

例

C++のモジュール定義

```
# include <boost/python/return_opaque_pointer.hpp>
# include <boost/python/def.hpp>
# include <boost/python/module.hpp>
# include <boost/python/return_value_policy.hpp>

typedef struct opaque_ *opaque;

opaque the_op    = ((opaque) 0x47110815);

opaque get () { return the_op; }
void use (opaque op) {
    if (op != the_op)
```

```
     throw std::runtime_error (std::string ("failed"));
}

void failuse (opaque op) {
    if (op == the_op)
     throw std::runtime_error (std::string ("success"));
}

BOOST_PYTHON_OPAQUE_SPECIALIZED_TYPE_ID(opaque_)

namespace bpl = boost::python;

BOOST_PYTHON_MODULE(opaque_ext)
{
    bpl::def (
        "get", &::get, bpl::return_value_policy<bpl::return_opaque_pointer>());
    bpl::def ("use", &::use);
    bpl::def ("failuse", &::failuse);
}
```

Python のコード

```
"""
>>> from opaque_ext import *
>>> #
>>> # 正しい変換のチェック
>>> use(get())
>>> failuse(get())
Traceback (most recent call last):
        ...
RuntimeError: success
>>> #
>>> # 整数から不透明なオブジェクトへの変換が存在しないことのチェック
>>> use(0)
Traceback (most recent call last):
        ...
TypeError: bad argument type for built-in operation
>>> #
>>> # 文字列から不透明なオブジェクトへの変換が存在しないことのチェック
>>> use("")
Traceback (most recent call last):
        ...
TypeError: bad argument type for built-in operation
"""
def run(args = None):
    import sys
    import doctest

    if args is not None:
        sys.argv = args
    return doctest.testmod(sys.modules.get(__name__))

if __name__ == '__main__':
    print "実行中..."
    import sys
    sys.exit(run()[0])
```

参照

opaque

Python との型変換

ヘッダ<boost/python/extract.hpp>

はじめに

一般的な Python オブジェクトから C++オブジェクトの値を抽出する機構をエクスポートする。`extract<...>`は object を特定の ObjectWrapper に「ダウンキャスト」するのにも使用できるということに注意していただきたい。可変の Python 型について同じ型で呼び出すと（例えば `list([1,2])`）一般的にはその引数の**コピー**が作成されるため、これが元のオブジェクトにおける ObjectWrapper インターフェイスにアクセスする唯一の方法となる可能性がある。

クラス

extract クラステンプレート

`extract<T>`を使用すると object のインスタンスから任意の C++型の値を抽出できる。2 つの使用方法をサポートする:

1. `extract<T>(o)`は、`T` へ暗黙に変換可能な一時オブジェクトである（オブジェクトの関数呼び出し演算子による明示的な変換も可能である）。しかしながら `o` から型 `T` のオブジェクトへの変換が利用できない場合は、Python の `TypeError` 例外を送出する。
2. `extract<T> x(o);`は、例外を投げることなく変換が可能か問い合わせる `check()` メンバ関数を持つ抽出子を構築する。

extract クラステンプレートの概要

```
namespace boost { namespace python
{
  template <class T>
  struct extract
  {
      typedef unspecified result_type;

      extract(PyObject*);
      extract(object const&);

      result_type operator()() const;
      operator result_type() const;

      bool check() const;
  };
}}
```

extract クラスのコンストラクタおよびデストラクタ

```
extract(PyObject* p);
extract(object const&);
```

要件

第 1 形式では *p* は非 null でなければならない。

効果

コンストラクタの引数が管理する Python オブジェクトへのポインタを格納する。特にオブジェクトの参照カウントは増加しない。抽出子の変換関数が呼び出される前にオブジェクトが破壊されないようにするのはユーザの責任である。

extract クラスのオブザーバ関数

```
result_type operator()() const;
operator result_type() const;
```

効果

格納したポインタを *result_type* へ変換する。これは `T` か `T const&` である。

戻り値

格納したポインタが参照するものに対応する *result_type* のオブジェクト。

例外

そのような変換が不可能な場合は [error_already_set](#)（`TypeError` を設定する）。実際に使用している変換器が未規定の他の例外を投げる可能性がある。

```
bool check() const;
```

事後条件

なし。特に戻り値が `true` であっても `operator result_type()` か `operator()()` が例外を投げないとは限らないことに注意していただきたい。

戻り値

格納したポインタから `T` への変換が不可能な場合**のみ** `false`。

例

```
#include <cstdio>
using namespace boost::python;
int Print(str s)
{
   // Python の文字列オブジェクトから C の文字列を抽出する
   char const* c_str = extract<char const*>(s);
```

```
   // printf で印字する
   std::printf("%s\n", c_str);

   // Python の文字列の長さを取得し、整数へ変換する
   return extract<int>(s.attr("__len__")())
}
```

以下は extract<...>と class_<...>を使用して、ラップした C++クラスのインスタンスを作成しアクセスする例である。

```
struct X
{
   X(int x) : v(x) {}
   int value() { return v; }
 private:
   int v;
};

BOOST_PYTHON_MODULE(extract_ext)
{
    object x_class(
       class_<X>("X", init<int>())
          .def("value", &X::value))
          ;

    // Python のインターフェイスを介して X のオブジェクトをインスタンス化する。
    // 寿命は以降、x_obj が管理する。
    object x_obj = x_class(3);

    // Python のオブジェクトを使用せずに C++オブジェクトへの参照を取得する
    X& x = extract<X&>(x_obj);
    assert(x.value() == 3);
}
```

ヘッダ<boost/python/implicit.hpp>

はじめに

implicitly_convertible は、Python オブジェクトを C++引数型に対して照合するとき、C++の暗黙・明示的な変換について暗黙的な利用を可能にする。

関数

implicitly_convertible 関数テンプレート

```
template <class Source, class Target>
void implicitly_convertible();
```

引数	説明
Source	暗黙の変換における元の型
Target	暗黙の変換における対象の型

要件

宣言 `Target t(s);`が合法である（`s`は*Source*型）。

効果

*Source*のrvalueを生成する登録済み変換器が1つでも存在する場合、あらゆる`PyObject* p`について変換が成功する*Target* rvalueへの`from_python`変換器を登録する。

根拠

C++ユーザは、C++で行っているような相互運用性をPythonで利用できると考える。

例

C++のモジュール定義

```
#include <boost/python/class.hpp>
#include <boost/python/implicit.hpp>
#include <boost/python/module.hpp>

using namespace boost::python;

struct X
{
    X(int x) : v(x) {}
    operator int() const { return v; }
    int v;
};

int x_value(X const& x)
{
    return x.v;
}

X make_x(int n) { return X(n); }

BOOST_PYTHON_MODULE(implicit_ext)
{
    def("x_value", x_value);
    def("make_x", make_x);

    class_<X>("X",
        init<int>())
        ;

    implicitly_convertible<X,int>();
    implicitly_convertible<int,X>();
}
```

Python のコード

```
>>> from implicit_ext import *
>>> x_value(X(42))
42
>>> x_value(42)
42
>>> x = make_x(X(42))
>>> x_value(x)
42
```

ヘッダ<boost/python/lvalue_from_pytype.hpp>

はじめに

<boost/python/lvalue_from_pypytype.hpp>は、与えられた型の Python インスタンスから C++オブジェクトを抽出する機能を提供する。典型的には、「伝統的な」Python の拡張型を取り扱う場合に有用である。

クラス

lvalue_from_pytype クラステンプレート

lvalue_from_pytype クラステンプレートはfrom_python変換器を登録する。この変換器は与えられた Python 型のオブジェクトから個々の C++型への参照およびポインタを抽出できる。テンプレート引数は次のとおり。

以下の表において *x* は PythonObject&型のオブジェクトである。[20]

引数	要件	セマンティクス
Extractor	参照型を返す execute 関数を持つ Extractor モデル。	Python オブジェクトから lvalue を抽出する（オブジェクトの型が適合していれば）。
python_type	PyTypeObject*コンパイル時定数。	この変換器が変換可能なインスタンスの Python 型。Python の派生型もまた変換可能である。

lvalue_from_pytype クラステンプレートの概要

```
namespace boost { namespace python
{
   template <class Extractor, PyTypeObject const* python_type>
   struct lvalue_from_pytype
   {
       lvalue_from_pytype();
   };
}}
```

20 訳注 *x* も PythonObject も見当たりませんが…。

lvalue_from_pytype クラステンプレートのコンストラクタ

```
lvalue_from_pytype();
```

効果

与えられた型の Python オブジェクトを `Extractor::execute` が返す型の lvalue へ変換する変換器を登録する。

extract_identity クラステンプレート

`extract_identity` は、抽出する C++型と Python オブジェクト型が同一であるありふれた場合に使用する Extractor モデルである。

extract_identity クラステンプレートの概要

```
namespace boost { namespace python
{
   template <class InstanceType>
   struct extract_identity
   {
      static InstanceType& execute(InstanceType& c);
   };
}}
```

extract_identity クラステンプレートの静的関数

```
InstanceType& execute(InstanceType& c);
```

戻り値

c

extract_member クラステンプレート

`extract_member` は、抽出する C++型が Python オブジェクトのメンバであるありふれた場合に使用する Extractor モデルである。

extract_member クラステンプレートの概要

```
namespace boost { namespace python
{
   template <class InstanceType, class MemberType, MemberType (InstanceType::*member)>
   struct extract_member
   {
      static MemberType& execute(InstanceType& c);
   };
}}
```

extract_member クラステンプレートの静的関数

```
static MemberType& execute(InstanceType& c);
```

戻り値

c.*member

例

以下の例では、Python のドキュメントにある標準的な noddy モジュール例を実装したとして、Noddy を操作するモジュールをビルドしたいとする。noddy_NoddyObject は特に気を引くような情報を持たない単純なものであるので、この例は少しばかりわざとらしい（何らかの理由で、特定の 1 つのオブジェクトに対する追跡を維持したいものとする）。このモジュールは noddy_NoddyType を定義するモジュールに動的にリンクしなければならない。

C++のモジュール定義

```
#include <boost/python/module.hpp>
#include <boost/python/handle.hpp>
#include <boost/python/borrowed.hpp>
#include <boost/python/lvalue_from_pytype.hpp>

// Python のドキュメントから引っ張り出した定義
typedef struct {
   PyObject_HEAD
} noddy_NoddyObject;

using namespace boost::python;
static handle<noddy_NoddyObject> cache;

bool is_cached(noddy_NoddyObject* x)
{
   return x == cache.get();
}

void set_cache(noddy_NoddyObject* x)
{
   cache = handle<noddy_NoddyObject>(borrowed(x));
}

BOOST_PYTHON_MODULE(noddy_cache)
{
   def("is_cached", is_cached);
   def("set_cache", set_cache);

   // Noddy の lvalue を扱う変換器
   lvalue_from_pytype<extract_identity<noddy_NoddyObject>,&noddy_NoddyType>();
}
```

Python のコード

```
>>> import noddy
>>> n = noddy.new_noddy()
>>> import noddy_cache
>>> noddy_cache.is_cached(n)
0
>>> noddy_cache.set_cache(n)
>>> noddy_cache.is_cached(n)
1
>>> noddy_cache.is_cached(noddy.new_noddy())
0
```

ヘッダ<boost/python/opaque_pointer_converter.hpp>

クラス

opaque<P>クラステンプレート

opaque<>は、自身を Python オブジェクトと未定義型へのポインタの双方向変換器として登録する。

opaque<P>クラステンプレートの概要

```
namespace boost { namespace python
{
    template<class Pointee>
    struct opaque
    {
        opaque();
    };
}}
```

opaque クラステンプレートのコンストラクタ

```
opaque();
```

効果

- Python オブジェクトから不透明なポインタへの lvalue_from_pytype 変換器としてインスタンスを登録する。作成される Python オブジェクトは、ラップする不透明なポインタが指す型の後ろに配置される。
- 不透明なポインタから Python オブジェクトへの to_python_converter としてインスタンスを登録する。

他のモジュールで登録されたインスタンスが既にある場合は、多重登録の警告を避けるため、このインスタンスは登録を再試行することはない。

注意

通常、このクラスのインスタンスは各 *Pointee* につき 1 つだけ作成する。

マクロ

BOOST_PYTHON_OPAQUE_SPECIALIZED_TYPE_ID(Pointee)

このマクロは、不完全型であるためインスタンス化が不可能な `type_id` 関数の特殊化を定義するのに使用しなければならない。

注意

不透明な変換器を使用する各翻訳単位でこのマクロを呼び出さなければならない。

参照

`return_opaque_pointer`

ヘッダ<boost/python/to_python_converter.hpp>

はじめに

`to_python_converter` は与えられた C++型のオブジェクトから Python オブジェクトへの変換を登録する。

クラス

to_python_converter クラステンプレート

`to_python_converter` は、第 2 テンプレート引数の静的メンバ関数に対するラッパを追加し、変換器のレジストリへの挿入といった低水準の詳細を処理する。

以下の表において *x* は `T` 型のオブジェクト。

引数	要件	説明
T		変換元オブジェクトの C++型。
Conversion	`p == 0` かつ `PyErr_Occurred() != 0` の場合、`PyObject* p = Conversion::convert(x)`	実際の変換を行う `convert` 静的メンバ関数を持つクラス型。
`bool` *has_get_pytype* `= false`	`PyTypeObject const * p = Conversion::get_pytype()`	**省略可能なメンバ。** *Conversion* が `get_pytype` を持つ場合、この引数に対して `true` を与えなければならない。この引数が与えられた場合、`get_pytype` はこの変換を使用する関数の戻り値の型に対してドキュメントを生成するために使用される。`get_pytype` は `pytype_function.hpp` のクラスと関数を使用して実装してもよい。注意：後方互換性のため、この引数を渡す

引数	要件	説明
		前に BOOST_PYTHON_SUPPORTS_PY_SIGNATURES が定義されているチェックするとよい（ここを見よ）。

to_python_converter クラステンプレートの概要

```
namespace boost { namespace python
{
  template <class T, class Conversion, bool convertion_has_get_pytype_member=false>
  struct to_python_converter
  {
      to_python_converter();
  };
}}
```

to_python_converter クラステンプレートのコンストラクタ

```
to_python_converter();
```

効果

Conversion::convert()を実際の動作として使用する to_python 変換器を登録する。

例

以下の例では、Python のドキュメントにある標準的な noddy モジュール例を実装したとして、関連する宣言を noddy.h に置いたものと仮定する。noddy_NoddyObject は極限なまでに単純な拡張型であるので、この例は少しばかりわざとらしい。すべての情報がその戻り値の**型**に含まれる関数をラップしている。

C++のモジュール定義

```
#include <boost/python/reference.hpp>
#include <boost/python/module.hpp>
#include "noddy.h"

struct tag {};
tag make_tag() { return tag(); }

using namespace boost::python;

struct tag_to_noddy
{
    static PyObject* convert(tag const& x)
    {
        return PyObject_New(noddy_NoddyObject, &noddy_NoddyType);
    }
    static PyTypeObject const* get_pytype()
    {
```

```
        return &noddy_NoddyType;
    }
};

BOOST_PYTHON_MODULE(to_python_converter)
{
    def("make_tag", make_tag);
    to_python_converter<tag, tag_to_noddy, true>(); // tag_to_noddy がメンバ get_pytype を持つので
「true」
}
```

Python のコード

```
>>> import to_python_converter
>>> def always_none():
...     return None
...
>>> def choose_function(x):
...     if (x % 2 != 0):
...         return to_python_converter.make_tag
...     else:
...         return always_none
...
>>> a = [ choose_function(x) for x in range(5) ]
>>> b = [ f() for f in a ]
>>> type(b[0])
<type 'NoneType'>
>>> type(b[1])
<type 'Noddy'>
>>> type(b[2])
<type 'NoneType'>
>>> type(b[3])
<type 'Noddy'>
```

ヘッダ<boost/python/register_ptr_to_python.hpp>

はじめに

<boost/python/register_ptr_to_python.hpp>は、スマートポインタから Python への変換を登録する関数テンプレート register_ptr_to_python を提供する。結果の Python オブジェクトは変換したスマートポインタのコピーを保持するが、参照先のラップ済みコピーであるかのように振舞う。参照先の型が仮想関数を持ちそのクラスが動的（最派生）型を表現する場合、Python のオブジェクトは最派生型に対するラッパのインスタンスとなる。1 つの参照先クラスに対して 2 つ以上のスマートポインタ型を登録可能である。

Python の X オブジェクトを smart_ptr<X>&（非 const 参照）へ変換するため、組み込む C++オブジェクトは smart_ptr<X>が保持しなければならない。またラップしたオブジェクトを Python からコンストラクタを呼び出して作成するとき、どのように保持するかは class_<...>インスタンスの HeldType 引数で決められることに注意していただきたい。

関数

```
template <class P>
void register_ptr_to_python()
```

要件

P が Dereferenceable。

効果

P のインスタンスの Python への変換を可能にする。

例

C++のラッパコード

以下の例は仮想関数を持つクラス A と、boost::shared_ptr<A>を扱う関数を持つモジュールである。

```
struct A
{
    virtual int f() { return 0; }
};

shared_ptr<A> New() { return shared_ptr<A>( new A() ); }

int Ok( const shared_ptr<A>& a ) { return a->f(); }

int Fail( shared_ptr<A>& a ) { return a->f(); }

struct A_Wrapper: A
{
    A_Wrapper(PyObject* self_): self(self_) {}
    int f() { return call_method<int>(self, "f"); }
    int default_f() { return A::f(); }
    PyObject* self;
};

BOOST_PYTHON_MODULE(register_ptr)
{
    class_<A, A_Wrapper>("A")
        .def("f", &A::f, &A_Wrapper::default_f)
    ;

    def("New", &New);
    def("Ok", &Call);
    def("Fail", &Fail);

    register_ptr_to_python< shared_ptr<A> >();
}
```

Python のコード

```
>>> from register_ptr import *
>>> a = A()
>>> Ok(a)      # OK、shared_ptr<A>として渡した
0
>>> Fail(a)    # shared_ptr<A>&として渡してしまった（Python 内で作成したのだった！）
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: bad argument type for built-in operation
>>>
>>> na = New()    # ここで "na" は実際は shared_ptr<A>
>>> Ok(a)
0
>>> Fail(a)
0
>>>
```

shared_ptr<A>を以下のように登録したとすると、

```
  class_<A, A_Wrapper, shared_ptr<A> >("A")
       .def("f", &A::f, &A_Wrapper::default_f)
   ;
```

shared_ptr<A>を shared_ptr<A_Wrapper>に変換しようとするとエラーになる。

```
>>> a = New()
Traceback (most recent call last):
File "<stdin>", line 1, in ?
TypeError: No to_python (by-value) converter found for C++ type: class boost::shared_ptr<struct A>
>>>
```

組み込み

ヘッダ<boost/python/exec.hpp>

はじめに

Python のインタープリタを C++コードへ組み込む機構をエクスポートする。

関数

eval

```
object eval(str expression,
```

```
            object globals = object(),
            object locals = object());
```

効果

辞書 *globals* および *locals* が指定した文脈で Python の式 *expression* を評価する。

戻り値

式の値を保持する object のインスタンス。

exec

```
object exec(str code,
            object globals = object(),
            object locals = object());
```

効果

辞書 *globals* および *locals* が指定した文脈で Python のソースコード *code* を実行する。

戻り値

コードの実行結果を保持する object のインスタンス。

exec_file

```
object exec_file(str filename,
                 object globals = object(),
                 object locals = object());
```

効果

辞書 *globals* および *locals* が指定した文脈で、*filename* で与えた名前のファイルから Python のソースコードを実行する。

戻り値

コードの実行結果を保持する object のインスタンス。

例

以下の例は、`import` と `exec` を使用して Python で関数を定義し、後で C++から呼び出している。

```
#include <iostream>
#include <string>

using namespace boost::python;

void greet()
{
  // main モジュールを得る。
  object main = import("__main__");

  // main モジュールの名前空間を得る。
```

```
  object global(main.attr("__dict__"));

  // Python 内で greet 関数を定義する。
  object result = exec(
    "def greet():                    \n"
    "    return 'Hello from Python!' \n",
    global, global);

  // greet 関数への参照を作成する。
  object greet = global["greet"];

  // 呼び出す。
  std::string message = extract<std::string>(greet());
  std::cout << message << std::endl;
}
```

文字列に Python のスクリプトを組み込む代わりに、ファイルに格納しておいてもよい。

```
def greet():
   return 'Hello from Python!'
```

代わりにこれを実行する。

```
  // ...
  // ファイルから greet 関数を読み込む。
  object result = exec_file(script, global, global);
  // ...
}
```

ヘッダ<boost/python/import.hpp>

はじめに

Python のモジュールをインポートする機構をエクスポートする。

関数

import

```
object import(str name);
```

効果

名前 *name* のモジュールをインポートする。

戻り値

インポートしたモジュールへの参照を保持する object のインスタンス。

例

以下の例は、import を使用して Python 内の関数にアクセスし、後で C++から呼び出している。

```
#include <iostream>
#include <string>

using namespace boost::python;

void print_python_version()
{
  // sys モジュールを読み込む。
  object sys = import("sys");

  // Python のバージョンを抽出する。
  std::string version = extract<std::string>(sys.attr("version"));
  std::cout << version << std::endl;
}
```

ユーティリティとインフラストラクチャ

ヘッダ<boost/python/has_back_reference.hpp>

はじめに

<boost/python/has_back_reference.hpp>は、述語メタ関数 has_back_reference<>を定義する。ユーザはこれを特殊化して、ラップするクラスのインスタンスが Python オブジェクトに対応する PyObject*を保持することを指定できる。

クラス

has_back_reference クラステンプレート

引数が pointer_wrapper<>である場合に value が真である単項メタ関数である。

has_back_reference クラステンプレートの概要

```
namespace boost { namespace python
{
    template<class WrappedClass> class has_back_reference
    {
        typedef mpl::false_ type;
    };
}}
```

ラップするクラスをどうように構築するか決定するため、Boost.Python がアクセスする「メタ関数」。

type::value は未規定型の論理値へ変換可能な整数定数である。class_<WrappedClass>::def(init<*type-*

sequence...>()）および暗黙のラップされたコピーコンストラクタ（noncopyable でない限り）の各呼び出しについて、対応するコンストラクタ WrappedClass::WrappedClass(PyObject*, *type-sequence*...)が存在する場合、true の値を持つ type の整数定数が特殊化により置換される可能性がある。そのような特殊化が存在する場合、WrappedClass のコンストラクタが Python から呼び出されるときは常に対応する Python オブジェクトへの「逆参照」ポインタが使用される。mpl::true_から特殊化を導出するのが、この入れ子の type を提供する最も簡単な方法である。

例

C++のモジュール定義

```
#include <boost/python/class.hpp>
#include <boost/python/module.hpp>
#include <boost/python/has_back_reference.hpp>
#include <boost/python/handle.hpp>
#include <boost/shared_ptr.hpp>

using namespace boost::python;
using boost::shared_ptr;

struct X
{
    X(PyObject* self) : m_self(self), m_x(0) {}
    X(PyObject* self, int x) : m_self(self), m_x(x) {}
    X(PyObject* self, X const& other) : m_self(self), m_x(other.m_x) {}

    handle<> self() { return handle<>(borrowed(m_self)); }
    int get() { return m_x; }
    void set(int x) { m_x = x; }

    PyObject* m_self;
    int m_x;
};

// Xについて has_back_reference を特殊化
namespace boost { namespace python
{
  template <>
  struct has_back_reference<X>
    : mpl::true_
  {};
}}

struct Y
{
    Y() : m_x(0) {}
    Y(int x) : m_x(x) {}
    int get() { return m_x; }
    void set(int x) { m_x = x; }

    int m_x;
};

shared_ptr<Y>
Y_self(shared_ptr<Y> self) { return self; }
```

```
BOOST_PYTHON_MODULE(back_references)
{
    class_<X>("X")
       .def(init<int>())
       .def("self", &X::self)
       .def("get", &X::get)
       .def("set", &X::set)
       ;

    class_<Y, shared_ptr<Y> >("Y")
       .def(init<int>())
       .def("get", &Y::get)
       .def("set", &Y::set)
       .def("self", Y_self)
       ;
}
```

以下の Python セッションでは、`x.self()`が何度呼び出しても毎回同じ Python オブジェクトを返すいっぽうで、`y.self()`は同じ`Y`インスタンスを参照する新しい Python オブジェクトを作成する。

Python のコード

```
>>> from back_references import *
>>> x = X(1)
>>> x2 = x.self()
>>> x2 is x
1
>>> (x.get(), x2.get())
(1, 1)
>>> x.set(10)
>>> (x.get(), x2.get())
(10, 10)
>>>
>>>
>>> y = Y(2)
>>> y2 = y.self()
>>> y2 is y
0
>>> (y.get(), y2.get())
(2, 2)
>>> y.set(20)
>>> (y.get(), y2.get())
(20, 20)
```

ヘッダ<boost/python/instance_holder.hpp>

はじめに

<boost/python/instance_holder.hpp>は、ラップするクラスの C++インスタンスを保持する型の基底クラスである`class instance_holder`を提供する。

クラス

instance_holder クラス

instance_holder は、その具象派生クラスが Python のオブジェクトラッパ内で C++クラスインスタンスを保持する抽象基底クラスである。Python 内で C++クラスラッパからの多重継承を可能にするために、そのような各 Python オブジェクトは数珠繋ぎの instance_holder を持つ。ラップする C++クラスの__init__関数が呼び出されると新しい instance_holder インスタンスが作成され、その install() 関数を使用して Python オブジェクトにインストールされる。instance_holder の各具象派生クラスは、保持する型について Boost.Python が問い合わせできるよう holds() の実装を提供しなければならない。保持する型をラップするコンストラクタをサポートするため、クラスは所有する Python オブジェクトを第 1 引数 PyObject*として受け取り、残りの引数を保持する型のコンストラクタに転送するコンストラクタも提供しなければならない。第 1 引数は Python 内で仮想関数をオーバーライド可能にするのに必要であり、instance_holder 派生クラスによっては無視される可能性がある。

instance_holder クラスの概要

```
namespace boost { namespace python
{
  class instance_holder : noncopyable
  {
   public:
      // デストラクタ
      virtual ~instance_holder();

      // instance_holder の変更関数
      void install(PyObject* inst) throw();

      // instance_holder のオブザーバ
      virtual void* holds(type_info) = 0;
  };
}}
```

instance_holder クラスのデストラクタ

```
virtual ~instance_holder();
```

効果

オブジェクトを破壊する。

instance_holder クラスの変更関数

```
void install(PyObject* inst) throw();
```

要件

inst が、ラップする C++クラス型かその派生型の Python のインスタンスである。

効果

新しいインスタンスを、保持するインスタンスの Python オブジェクトの数珠繋ぎの先頭にインストールする。

例外

なし。

instance_holder クラスのオブザーバ

```
virtual void* holds(type_info x) = 0;
```

戻り値

`*this` がオブジェクトを保持している場合、*x* が示す型のオブジェクトへのポインタ。それ以外の場合 0。

例

以下の例は、Boost.Python がスマートポイントが保持するクラスをラップするのに使用しているインスタンスホルダテンプレートの簡易バージョンである。

```
template <class SmartPtr, class Value>
struct pointer_holder : instance_holder
{
   // SmartPtr 型から構築する
   pointer_holder(SmartPtr p)
       :m_p(p)

   // 保持する型の転送コンストラクタ
   pointer_holder(PyObject*)
       :m_p(new Value())
   {
   }

   template<class A0>
   pointer_holder(PyObject*,A0 a0)
       :m_p(new Value(a0))
   {
   }

   template<class A0,class A1>
   pointer_holder(PyObject*,A0 a0,A1 a1)
       :m_p(new Value(a0,a1))
   {
   }
   ...

 private: // ホルダに必要な実装
   void* holds(type_info dst_t)
   {
       // SmartPtr 型のインスタンスと...
       if (dst_t == python::type_id<SmartPtr>())
```

```
            return &this->m_p;

        // ...SmartPtr の element_type インスタンス
        // （ポインタが非 null の場合）を保持する
        return python::type_id<Value>() == dst_t ? &*this->m_p : 0;
    }

 private: // データメンバ
    SmartPtr m_p;
};
```

ヘッダ<boost/python/pointee.hpp>

はじめに

<boost/python/pointee.hpp>は、ポインタやスマートポインタの型から「ポイントされている」型を抽出する特性メタ関数テンプレート pointee<T>を導入する。

クラス

pointee<class T>クラステンプレート

pointee<T>は、class_<...>テンプレートでポインタやスマートポインタ型を HeldType 引数に使用するときに保持する型を推論するのに使用する。

pointee クラステンプレートの概要

```
namespace boost { namespace python
{
    template <class T> struct pointee
    {
        typedef T::element_type type;
    };

    // ポインタに対する特殊化
    template <T> struct pointee<T*>
    {
        typedef T type;
    };
}
```

例

サードパーティ製のスマートポインタ型 smart_pointer<T>があるとして、pointee<smart_pointer<T> >を部分特殊化してクラスラッパの HeldType として使用可能にする。

```
#include <boost/python/pointee.hpp>
#include <boost/python/class.hpp>
#include <third_party_lib.hpp>

namespace boost { namespace python
{
  template <class T> struct pointee<smart_pointer<T> >
  {
     typedef T type;
  };
}}

BOOST_PYTHON_MODULE(pointee_demo)
{
   class_<third_party_class, smart_pointer<third_party_class> >("third_party_class")
      .def(...)
      ...
      ;
}
```

ヘッダ<boost/python.hpp>

はじめに

Boost.Python ライブラリの公開インターフェイスヘッダをすべてインクルードする便利なヘッダである。

```
# include <args.hpp>
# include <args_fwd.hpp>
# include <back_reference.hpp>
# include <bases.hpp>
# include <borrowed.hpp>
# include <call.hpp>
# include <call_method.hpp>
# include <class.hpp>
# include <copy_const_reference.hpp>
# include <copy_non_const_reference.hpp>
# include <data_members.hpp>
# include <def.hpp>
# include <default_call_policies.hpp>
# include <dict.hpp>
# include <enum.hpp>
# include <errors.hpp>
# include <exception_translator.hpp>
# include <extract.hpp>
# include <handle.hpp>
# include <has_back_reference.hpp>
# include <implicit.hpp>
# include <init.hpp>
# include <instance_holder.hpp>
# include <iterator.hpp>
# include <list.hpp>
# include <long.hpp>
# include <lvalue_from_pytype.hpp>
# include <make_function.hpp>
# include <manage_new_object.hpp>
# include <module.hpp>
```

```
# include <numeric.hpp>
# include <object.hpp>
# include <object_protocol.hpp>
# include <object_protocol_core.hpp>
# include <operators.hpp>
# include <other.hpp>
# include <overloads.hpp>
# include <pointee.hpp>
# include <ptr.hpp>
# include <reference_existing_object.hpp>
# include <return_internal_reference.hpp>
# include <return_value_policy.hpp>
# include <scope.hpp>
# include <self.hpp>
# include <slice_nil.hpp>
# include <str.hpp>
# include <to_python_converter.hpp>
# include <to_python_indirect.hpp>
# include <to_python_value.hpp>
# include <tuple.hpp>
# include <type_id.hpp>
# include <with_custodian_and_ward.hpp>
```

ヘッダ<boost/python/handle.hpp>

はじめに

<boost/python/handle.hpp>は参照カウント付きの Python オブジェクトを管理するスマートポインタである class template handle を提供する。

クラス

handle クラステンプレート

handle は Python のオブジェクト型へのスマートポインタであり、T*型のポインタを保持する（T はそのテンプレート引数）。T は PyObject の派生型か、先頭 sizeof(PyObject)バイトが PyObject とレイアウト互換な POD 型のいずれかでなければならない。Python/'C' API と高水準コードの境界で handle<>を使用することだ。一般的なインターフェイスに対しては Python のオブジェクトよりも object を使用すべきだ。

このドキュメントで「upcast」は、Y が T の派生型の場合 static_cast<T*>で、そうでない場合 C スタイルのキャストでポインタ Y*を基底クラスポインタ T*へ変換する操作を指す。しかしながら Y の先頭 sizeof(PyObject)バイトが PyObject とレイアウト互換でなければ、「upcast」は違法である。

handle クラステンプレートの概要

```
namespace boost { namespace python
{
```

```
  template <class T>
  class handle
  {
      typedef unspecified-member-function-pointer bool_type;

   public: // 型
      typedef T element_type;

   public: // メンバ関数
      ~handle();

      template <class Y>
      explicit handle(detail::borrowed<null_ok<Y> >* p);

      template <class Y>
      explicit handle(null_ok<detail::borrowed<Y> >* p);

      template <class Y>
      explicit handle(detail::borrowed<Y>* p);

      template <class Y>
      explicit handle(null_ok<Y>* p);

      template <class Y>
      explicit handle(Y* p);

      handle();

      handle& operator=(handle const& r);

      template<typename Y>
      handle& operator=(handle<Y> const & r); // 例外を投げない


      template <typename Y>
      handle(handle<Y> const& r);

      handle(handle const& r);

      T* operator-> () const;
      T& operator* () const;
      T* get() const;
      void reset();
      T* release();

      operator bool_type() const; // 例外を投げない
   private:
      T* m_p;
  };

  template <class T> struct null_ok;
  namespace detail { template <class T> struct borrowed; }
}}
```

handle クラステンプレートのコンストラクタおよびデストラクタ

```
virtual ~handle();
```

効果

Py_XDECREF(upcast<PyObject*>(m_p))

```
template <class Y>
explicit handle(detail::borrowed<null_ok<Y> >* p);
```

効果

Py_XINCREF(upcast<PyObject*>(p)); m_p = upcast<T*>(p);

```
template <class Y>
explicit handle(null_ok<detail::borrowed<Y> >* p);
```

効果

Py_XINCREF(upcast<PyObject*>(p)); m_p = upcast<T*>(p);

```
template <class Y>
explicit handle(detail::borrowed<Y>* p);
```

効果

Py_XINCREF(upcast<PyObject*>(p)); m_p = upcast<T*>(expect_non_null(p));

```
template <class Y>
explicit handle(null_ok<Y>* p);
```

効果

m_p = upcast<T*>(p);

```
template <class Y>
explicit handle(Y* p);
```

効果

m_p = upcast<T*>(expect_non_null(p));

```
handle();
```

効果

m_p = 0;

```
template <typename Y>
handle(handle<Y> const& r);
handle(handle const& r);
```

効果

```
m_p = r.m_p; Py_XINCREF(upcast<PyObject*>(m_p));
```

handle クラステンプレートの変更メソッド

```
handle& operator=(handle const& r);
template<typename Y>
handle& operator=(handle<Y> const & r); // 例外を投げない
```

効果

```
Py_XINCREF(upcast<PyObject*>(r.m_p)); Py_XDECREF( upcast<PyObject*>(m_p)); m_p = r.m_p;
```

```
T* release();
```

効果

```
T* x = m_p; m_p = 0; return x;
```

```
void reset();
```

効果

```
*this = handle<T>();
```

handle クラスのオブザーバ関数

```
T* operator-> () const;
T* get() const;
```

戻り値

```
m_p;
```

```
T& operator* () const;
```

戻り値

```
*m_p;
```

```
operator bool_type() const; // 例外を投げない
```

戻り値

`m_p == 0` の場合 0。それ以外の場合、`true` へ変換可能なポインタ。

関数

borrowed

```
template <class T>
detail::borrowed<T>* borrowed(T* p)
{
    return (detail::borrowed<T>*)p;
}
```

allow_null

```
template <class T>
null_ok<T>* allow_null(T* p)
{
    return (null_ok<T>*)p;
}
```

ヘッダ<boost/python/type_id.hpp>

はじめに

<boost/python/type_id.hpp>は、<typeinfo>のような実行時型識別のための型および関数を提供する。主にコンパイラのバグやプラットフォーム固有の共有ライブラリとの相互作用に対する回避策のために存在する。

クラス

type_info クラス

type_info インスタンスは型を識別する。std::type_info が規定しているとおり（ただしコンパイラによっては異なる実装をしている場合もある）、boost::python::type_info はトップレベルの参照や CV 指定子を表現しない（C++標準の 5.2.8 節を見よ）。std::type_info と異なり boost::python::type_info インスタンスはコピー可能であり、共有ライブラリ境界をまたいで確実に動作する。

type_info クラスの概要

```
namespace boost { namespace python
{
  class type_info : totally_ordered<type_info>
  {
   public:
      // コンストラクタ
      type_info(std::type_info const& = typeid(void));
```

```
      // 比較
      bool operator<(type_info const& rhs) const;
      bool operator==(type_info const& rhs) const;

      // オブザーバ
      char const* name() const;
  };
}}
```

type_info クラスのコンストラクタ

```
type_info(std::type_info const& = typeid(void));
```

効果

引数と同じ型を識別する type_info オブジェクトを構築する。

根拠

type_info オブジェクトの配列の作成が必要になることがあるため、親切にも既定の引数が与えられている。

注意

このコンストラクタはコンパイラの typeid() 実装の非準拠を修正**しない**。以下の type_id を見よ。

type_info クラスの比較関数

```
bool operator<(type_info const& rhs) const;
```

効果

type_info オブジェクト間の全順序を与える。

```
bool operator==(type_info const& rhs) const;
```

戻り値

2 つの値が同じ型を示す場合は true。

注意

totally_ordered<type_info>を非公開基底クラスとして使用すると、<=、>=、>および!=が提供される。

type_info クラスのオブザーバ関数

```
char const* name() const;
```

戻り値

オブジェクトの構築に使用した引数に対して name() を呼び出した結果。

関数

operator<<

```
std::ostream& operator<<(std::ostream&s, type_info const&x);
```

効果

x が指定する型の説明を *s* に書き込む。

根拠

すべての C++実装が真に可読可能な `type_info::name()` 文字列を提供するわけではないが、文字列を復号化して手ごろな表現を生成できる場合がある。

type_id

```
template <class T> type_info type_id()
```

戻り値

`type_info(typeid(T))`

注意

標準に非準拠ないくつかの C++実装において、コードは実際には上記のように単純ではない。その C++実装が標準に準拠しているかのように動作するようセマンティクスを調整する。

例

以下の例は、多少醜いが `type_id` 機能の使用方法を示している。

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

// ユーザが int の引数を渡した場合に true を返す
template <class T>
bool is_int(T x)
{
   using boost::python::type_id;
   return type_id<T>() == type_id<int>();
}
```

ヘッダ<boost/python/ssize_t.hpp>

はじめに

Python 2.5 は新しい型定義 `Py_ssize_t` および 2 つの関連マクロを導入した（PEP 353）。`<boost/python/ssize_t.hpp>`ヘッダはこれらの定義を `ssize_t`、`ssize_t_max` および `ssize_t_min` として `boost::python` 名前空間にインポートする。後方互換

性のために、Python の以前バージョンでは適切な定義を提供する。

型定義

可能であれば Py_ssize_t を boost::python 名前空間にインポートする。または後方互換性のために適切な型定義を提供する。

```
#if PY_VERSION_HEX >= 0x02050000
typedef Py_ssize_t ssize_t;
#else
typedef int ssize_t;
#endif
```

定数

可能であれば PY_SSIZE_T_MAX および PY_SSIZE_T_MIN を boost::python 名前空間に定数としてインポートする。または後方互換性のために適切な定数を提供する。

```
#if PY_VERSION_HEX >= 0x02050000
ssize_t const ssize_t_max = PY_SSIZE_T_MAX;
ssize_t const ssize_t_min = PY_SSIZE_T_MIN;
#else
ssize_t const ssize_t_max = INT_MAX;
ssize_t const ssize_t_min = INT_MIN;
#endif
```

トピック

Python の関数とメソッドの呼び出し

はじめに

Python 関数を保持する object インスタンス f が与えられたとき、C++からその関数を呼び出す最も簡単な方法は、単にその関数呼び出し演算子を起動することである。

```
f("tea", 4, 2) // Python の f('tea', 4, 2) と同じ
```

また、当然のことながら object インスタンス x のメソッドを呼び出すには対応する属性の関数呼び出し演算子を使用する。

```
x.attr("tea")(4, 2); // Python の x.tea(4, 2) と同じ
```

object インスタンスがない場合、PyObject*に対して Python の関数およびメソッドを呼び出すために、Boost.Python は 2 種類の関数テンプレート call および call_method を提供している。Python の関数オブジェクト（または Python の呼び出し可能オブジェ

クト）を呼び出すインターフェイスは次のようなものである。

```
call<ResultType>(callable_object, a1, a2... aN);
```

Python オブジェクトのメソッド呼び出しも同様に簡単である。

```
call_method<ResultType>(self_object, "method-name", a1, a2... aN);
```

この比較的低水準なインターフェイスは、Python でオーバーライド可能な C++仮想関数を実装するときに使用する。

引数の処理

引数はその型に従って Python に変換する。既定では引数 a1...a*N* は新しい Python オブジェクトにコピーされるが、ptr() および ref() を使用してこの振る舞いを上書きすることができる。

```
class X : boost::noncopyable
{
   ...
};

void apply(PyObject* callable, X& x)
{
   // callable を呼び出し、x への参照を保持する Python オブジェクトを渡す
   boost::python::call<void>(callable, boost::ref(x));
}
```

以下の表において *x* は実際の引数オブジェクトであり、*cv* は省略可能な CV 指定子（const、volatile あるいは const volatile）である。

引数の型	振る舞い
T cv& T cv	T を返すラップした C++関数の戻り値と同じ方法で Python 引数を作成する。T がクラス型の場合、通常は x を新しい Python オブジェクト内にコピー構築する。
T*	x == 0 の場合、Python 引数は None である。それ以外の場合、T を返すラップする C++関数の戻り値と同じ方法で Python 引数を作成する。T がクラス型の場合、通常は *x を新しい Python オブジェクト内にコピー構築する。
boost::reference_wrapper<T>	Python の引数は（コピーではなく）x.get() へのポインタを持つ。注意：結果のオブジェクトへの参照を保持する Python コードが *x.get() の寿命を超えて存在しないようにしないと、**クラッシュする！**
pointer_wrapper<T>	x.get() == 0 の場合、Python 引数は None である。それ以外の場合、Python の引数は（コピーではなく）x.get() へのポインタを持つ。注意：結果のオブジェクトへの参照を保持する Python コードが *x.get() の寿命を超えて存在しないようにしないと、**クラッシュする！**

戻り値の処理

大抵の場合 `call<ResultType>()` および `call_method<ResultType>()` は、`ResultType` に対して登録したすべての lvalue および rvalue の `from_python` 変換器を利用し、結果のコピーである `ResultType` を返す。しかしながら `ResultType` がポインタか参照型の場合、Boost.Python は lvalue の変換器のみを探索する。懸垂ポインタおよび参照を避けるため、結果の Python オブジェクトの参照カウントが 1 の場合は例外を投げる。

根拠

通常 `a1`...`aN` に対応する Python の引数を得るには、それぞれについて新しい Python オブジェクトを作成しなければならない。C++オブジェクトをその Python オブジェクトにコピーすべきだろうか、あるいは Python オブジェクトが単に C++オブジェクトへの参照かポインタを保持すべきだろうか。大抵の場合、呼び出される関数はどこかに行ってしまった Python オブジェクトへの参照を保持する可能性があるため、後者の方法は安全ではない。C++オブジェクトを破壊した後に Python オブジェクトを使用すると、Python がクラッシュする。

Python 側のユーザがインタープリタのクラッシュについて気を払うべきでないという原理を踏まえ、既定の振る舞いは C++オブジェクトをコピーすることとなっており、コピーを行わない振る舞いはユーザが直接 `a1` と書く代わりに `boost::ref(a1)` とした場合のみ認められる。こうすることで、少なくとも「意図せず」危険な振る舞いを遭遇することはない。コピーを伴わない（「参照」による）振る舞いは通常、クラス型でのみ利用可能であり、それ以外で使用すると実行時に Python の例外を送出して失敗する[21]ことも付記しておく。

しかしながらポインタ型が問題となる。方法の 1 つはいずれかの `aN` がポインタ型である場合にコンパイルを拒絶することである。何といってもユーザは「値渡し」として `*aN` を渡すことができ、`ref(*aN)` で参照渡しの振る舞いを示すことができる。しかしこれでは null ポインタから `None` への変換で問題が起こる。null ポインタ値を参照剥がしすることは違法である。

折衷案として私が下した決断は以下のとおりだ:

1. 既定の振る舞いは値渡しとする。非 null ポインタを渡すと、参照先が新しい Python オブジェクトにコピーされる。それ以外の場合、対応する Python 引数は `None` である。
2. 参照渡しの振る舞いが必要な場合は、`aN` がポインタであれば `ptr(aN)` を、そうでなければ `ref(aN)` を使用する。`ptr(aN)` に null ポインタを渡すと、対応する Python 引数は `None` である。

戻り値についても類似の問題がある。`ResultType` にポインタ型か参照型を認めてしまうと、参照先のオブジェクトの寿命はおそらく Python オブジェクトに管理されることになる。この Python オブジェクトが破壊されるとポインタは懸垂する。`ResultType` が `char const*` の場合、特に問題は深刻である。対応する Python の String オブジェクトは一般に参照カウントが 1 であり、つまり `call<char const*>(...)` が返った直後にポインタは懸垂する。

以前の Boost.Python v1 は `call<char const*>()` のコンパイルを拒絶することでこの問題に対処したが、これは極端でありかつ何の解決にもならない。極端というのは、所有する Python の文字列オブジェクト（例えば Python のクラス名である場合）が呼び出しを超えて生存する可能性があるためである。また、他の型のポインタや参照の戻り値についてもまったく同様の問題があるため、結

21 `int` や `char` のような非クラス型についてはコンパイル時に失敗させることも可能だろうが、それがこの制限を課す優れた方法であるか私にはよく分からない。

局は解決にならないわけである。

Boost.Python v2 では次のように対処した。

1. コールバックの `const char*` 戻り値型に対するコンパイル時の制限を除去した。
2. `U` がポインタ型か参照型の場合、戻り値の Python オブジェクトの参照カウントが 1 である場合を検出し、`call<U>(...)` 内で例外を投げる。

ユーザは `call<U>` 内で `U` について明示的にポインタ、参照を指定しなければならないため安全であり、実行時の懸垂からも保護される。少なくとも `call<U>(...)` の呼び出しから抜け出すには十分である。

Boost.Python における pickle のサポート

pickle はオブジェクトの直列化（または永続化、整列化、平坦化）のための Python モジュールである。

オブジェクトの内容をファイルに保存、またはファイルから復元する必要があることはよくある。解法の 1 つは、特殊な形式でファイルへデータを読み書きする関数の組を書くことである。他の強力な解法は Python の pickle モジュールを使うことである。Python の自己記述機能を利用すると、pickle モジュールはほとんど任意の Python オブジェクトを再帰的にファイルに書き込み可能なバイトストリームへ変換する。

Boost.Python ライブラリは、Python ライブラリリファレンスの pickle の項に詳細記載のインターフェイスを通じて pickle モジュールをサポートする。このインターフェイスは以下に述べる特殊メソッド `__getinitargs__`、`__getstate__` および `__setstate__` を必要とする。Boost.Python は Python の cPickle モジュールとも完全に互換であることに注意していただきたい。

Boost.Python の pickle インターフェイス

ユーザレベルでは、Boost.Python の pickle インターフェイスは 3 つの特殊メソッドを伴う。

`__getinitargs__`	Boost.Python 拡張クラスのインスタンスを pickle 化するとき、pickler はインスタンスが `__getinitargs__` メソッドを持っているかテストする。このメソッドは Python のタプルを返さなければならない（`boost::python::tuple` を使うのが最も便利である）。インスタンスを unpickler が復元するとき、このタプルの内容をクラスのコンストラクタの引数として使用する。 `__getinitargs__` が定義されていない場合、`pickle.load` は引数無しでコンストラクタ（`__init__`）を呼び出す。すなわちオブジェクトはデフォルトコンストラクト可能でなければならない。
`__getstate__`	Boost.Python 拡張クラスのインスタンスを pickle 化するとき、pickler はインスタンスが `__getstate__` メソッドを持っているかテストする。このメソッドはインスタンスの状態を表す Python オブジェクトを返さなければならない。
`__setstate__`	Boost.Python 拡張クラスのインスタンスを unpickler により復元（`pickle.load`）するとき、はじめに `__getinitargs__` の結果を引数として構築する（上述）。次に unpickler は新しいインスタンスが `__setstate__` メソッドを持っているかテストする。テストが成功した場合、`__getstate__` の結果（Python オブジェクト）を引数としてこのメソッドを呼び出す。

上記 3 つの特殊メソッドは、ユーザが個別に `.def()` してもよい。しかしながら Boost.Python は簡単に使用できる高水準インター

フェイスを boost::python::pickle_suite クラスで提供している。このクラスは、__getstate__および__setstate__を組として定義しなければならないという一貫性も強制する。このインターフェイスの使用方法は以下の例で説明する。

例

boost/libs/python/test に、pickle サポートを提供する方法を示したファイルが 3 つある。

pickle1.cpp

この例の C++クラスは、コンストラクタに適切な引数を渡すことで完全に復元できる。よって pickle インターフェイスのメソッド__getinitargs__を定義するのに十分である。以下のようにする。

C++の pickle 関数の定義:

```
struct world_pickle_suite : boost::python::pickle_suite
{
  static
  boost::python::tuple
  getinitargs(world const& w)
  {
      return boost::python::make_tuple(w.get_country());
  }
};
```

Python の束縛を確立する。

```
class_<world>("world", args<const std::string&>())
    // ...
    .def_pickle(world_pickle_suite())
    // ...
```

pickle2.cpp

この例の C++クラスは、コンストラクタで復元不可能なメンバデータを持つ。よって pickle インターフェイスのメソッド組__getstate__、__setstate__を提供する必要がある。

C++の pickle 関数の定義:

```
struct world_pickle_suite : boost::python::pickle_suite
{
  static
  boost::python::tuple
  getinitargs(const world& w)
  {
    // ...
  }

  static
  boost::python::tuple
  getstate(const world& w)
  {
```

```
    // ...
  }

  static
  void
  setstate(world& w, boost::python::tuple state)
  {
    // ...
  }
};
```

suite 全体の Python の束縛を確立する。

```
class_<world>("world", args<const std::string&>())
    // ...
    .def_pickle(world_pickle_suite())
    // ...
```

簡単のために、__getstate__の結果に__dict__は含まれない。これは通常は推奨しないが、オブジェクトの__dict__が常に空であると分かっている場合は有効な方法である。この想定が崩れるケースは以下に述べる安全柵で捕捉できる。

pickle3.cpp

この例は pickle2.cpp と似ているが、__getstate__の結果にオブジェクトの__dict__が含まれる。より多くのコードが必要になるが、オブジェクトの__dict__が空とは限らない場合は避けられない。

落とし穴と安全柵

上述の pickle プロトコルには、Boost.Python 拡張モジュールのエンドユーザが気にかけない重大な落とし穴がある。

__getstate__が定義されており、インスタンスの__dict__が空でない

Boost.Python 拡張クラスの作成者は、以下の可能性を考慮せずに__getstate__を提供する可能性がある。

- クラスが Python 内で基底クラスとして使用される。おそらく派生クラスのインスタンスの__dict__は、インスタンスを正しく復元するために pickle 化する必要がある。
- ユーザがインスタンスの__dict__に直接要素を追加する。この場合もインスタンスの__dict__は pickle 化が必要である。

この高度に不明確な問題をユーザに警告するために、安全柵が提供されている。__getstate__が定義されており、インスタンスの__dict__が空でない場合は、Boost.Python はクラスが属性__getstate_manages_dict__を持っているかテストする。この属性が定義されていなければ例外を送出する。

```
RuntimeError: Incomplete pickle support (__getstate_manages_dict__ not set)
```

この問題を解決するには、まず__getstate__および__setstate__メソッドがインスタンスの__dict__を正しく管理するようにしなければならない。これは C++あるいは Python レベルのいずれでも達成可能であることに注意していただきたい。最後に安全柵

を故意にオーバライドしなければならない。例えばC++では以下のとおり（pickle3.cppから抜粋）。

```
struct world_pickle_suite : boost::python::pickle_suite
{
  // ...

  static bool getstate_manages_dict() { return true; }
};
```

あるいはPythonでは次のとおり。

```
import your_bpl_module
class your_class(your_bpl_module.your_class):
  __getstate_manages_dict__ = 1
  def __getstate__(self):
    # ここにコードを書く
  def __setstate__(self, state):
    # ここにコードを書く
```

実践的なアドバイス

- 多くの拡張クラスを持つBoost.Python拡張モジュールでは、すべてのクラスについてpickleの完全なサポートを提供すると著しいオーバーヘッドとなる。通常、完全なpickleサポートの実装は最終的にpickle化する拡張クラスに限定すべきである。
- インスタンスが__getinitargs__による再構築も可能な場合は__getstate__は避けよ。これは上記の落とし穴を自動的に避けることになる。
- __getstate__が必要な場合、返すPythonオブジェクトにインスタンスの__dict__を含めよ。

軽量な代替：Python側でのpickleサポートの実装

pickle4.cpp

pickle4.cppの例は、pickleサポートの実装に関する別のテクニックのデモンストレーションである。はじめにclass_::enable_pickling()メンバ関数でpickle化に必要な基本的な属性だけをBoost.Pythonに定義させる。

```
class_<world>("world", args<const std::string&>())
    // ...
    .enable_pickling()
    // ...
```

これでPythonのドキュメントにある標準的なPythonのpickleインターフェイスが有効になる。__getinitargs__メソッドをラップするクラス定義に「注入」することで、すべてのインスタンスをpickle化可能にする。

```
# ラップしたworldクラスをインポート
from pickle4_ext import world
```

```
# __getinitargs__の定義
def world_getinitargs(self):
  return (self.get_country(),)

# ここで__getinitargs__を注入（Pythonは動的言語！）
world.__getinitargs__ = world_getinitargs
```

Pythonから追加のメソッドを注入する方法については、チュートリアルの節も見よ。

添字アクセスのサポート

ヘッダ<boost/python/indexing/indexing_suite.hpp>

ヘッダ<boost/python/indexing/vector_indexing_suite.hpp>

はじめに

indexingは、添字アクセス可能（indexable）なC++コンテナをPythonへ容易にエクスポートするためのBoost.Pythonの機能である。添字アクセス可能なコンテナとは`operator[]`によりランダムアクセス可能なコンテナである（例：`std::vector`）。

`std::vector`のようなどこにでもある添字アクセス可能なC++コンテナをPythonへエクスポートするのに必要な機能はBoost.Pythonはすべて有しているが、その方法はあまり直感的ではない。Pythonのコンテナから C++コンテナへの変換は容易ではない。Pythonのコンテナを Boost.Pythonを使用して C++内でエミュレート（Pythonリファレンスマニュアルの「コンテナ型をエミュレートする」を見よ）するのは簡単ではない。C++コンテナを Pythonへ変換する以前に考慮すべきことが多数ある。これには`__len__`、`__getitem__`、`__setitem__`、`__delitem__`、`__iter__`および`__contains__`メソッドに対するラッパ関数の実装が含まれる。

目的は、

- 添字アクセス可能なC++コンテナの振る舞いを、Pythonコンテナの振る舞いに一致させる。
- `c[i]`が変更可能であるといった、コンテナ要素のインデクス（`__getitem__`）に対する既定の参照セマンティクスを提供する。要件は以下のとおり（`m`は非const（可変）メンバ関数（メソッド））。

```
val = c[i]
c[i].m()
val == c[i]
```

- `__getitem__`から安全な参照を返す：後でコンテナに追加、コンテナから削除しても懸垂参照が発生しない（Pythonをクラッシュさせない）。
- スライスの添字をサポート。
- 適切な場合はPythonのコンテナ引数（例：`list`、`tuple`）を受け付ける。
- 再定義可能なポリシークラスによる拡張性。
- ほとんどのSTLおよびSTLスタイルの添字アクセス可能なコンテナに対する定義済みサポートを提供する。

Boost.Python の indexing インターフェイス

indexing_suite［ヘッダ<boost/python/indexing/indexing_suite.hpp>］

`indexing_suite` クラスは、Python に調和させる C++コンテナを管理するための基底クラスである。目的は C++コンテナの外観と振る舞いを Python コンテナのそれに一致させることである。このクラスは自動的に（Python リファレンスの「コンテナ型をエミュレートする」の）Python の特殊メソッドをラップする。

__len__(*self*)	組み込み関数 `len()` を実装するために呼び出される。オブジェクトの長さ（0 以上の整数）を返さなければならない。また `__nonzero__()` メソッドを定義せず `__len__()` メソッドが 0 を返すオブジェクトは、論理値の文脈で偽として扱われる。
__getitem__(*self*, *key*)	`self[key]` の評価を実装するために呼び出される。シーケンス型では、受け取るキーは整数およびスライスオブジェクトでなければならない。負の添字に対する特殊な解釈（クラスがシーケンス型をエミュレートする場合）は `__getitem__()` メソッドの仕事であることに注意していただきたい。*key* が不適な型な場合は `TypeError` を送出し、値が（負の値に対する特殊な解釈の後）シーケンスの添字の集合外である場合は `IndexError` を送出しなければならない。注意：シーケンスの終了を適切に検出するため、`for` ループは不正な添字に対して `IndexError` が送出することを想定している。
__setitem__(*self*, *key*, *value*)	`self[key]` への代入を実装するために呼び出される。注意すべき点は `__getitem__()` と同じである。このメソッドは、マップ型についてはオブジェクトがキーに対する値の変更をサポートするか新しいキーを追加可能な場合、シーケンス型については要素が置換可能な場合のみ実装すべきである。不適な *key* 値に対しては `__getitem__()` メソッドと同様の例外を送出しなければならない。
__delitem__(*self*, *key*)	`self[key]` の削除を実装するために呼び出される。注意すべき点は `__getitem__()` と同じである。このメソッドは、マップ型についてはオブジェクトがキーの削除をサポートする場合、シーケンス型については要素をシーケンスから削除可能な場合のみ実装すべきである。不適な *key* 値に対しては `__getitem__()` メソッドと同様の例外を送出しなければならない。
__iter__(*self*)	このメソッドは、コンテナに対してイテレータが要求されたときに呼び出される。このメソッドは、コンテナ内のすべてのオブジェクトを走査する新しいイテレータオブジェクトを返さなければならない。マップ型については、コンテナのキーを走査しなければならず、`iterkeys()` メソッドとしても利用可能でなければならない。 イテレータオブジェクトもまたこのメソッドを実装する必要があり、自分自身を返さなければならない。イテレータオブジェクトの詳細については、Python ライブラリリファレンスの「イテレータ型」を見よ。
__contains__(*self*, *item*)	メンバ関係テスト操作を実装するために呼び出される。*self* 内に *item* があれば真を、そうでなければ偽を返さなければならない。マップ型オブジェクトについては、値やキー・値の組ではなくキーを対象とすべきである。

indexing_suite の派生クラス

indexing_suite はそのままで使用することを意図していない。indexing_suite の派生クラスにより 2、3 のポリシー関数を提供しなければならない。しかしながら、標準的な添字アクセス可能な STL コンテナのための indexing_suite 派生クラス群が提供されている。ほとんどの場合、単に定義済みのクラス群を使用すればよい。場合によっては必要に応じて定義済みクラスを改良してもよい。

vector_indexing_suite［ヘッダ<boost/python/indexing/vector_indexing_suite.hpp>］

vector_indexing_suite クラスは、std::vector クラス（および std::vector スタイルのクラス（例：std::vector のインターフェイスを持つクラス））をラップするために設計された定義済みの indexing_suite 派生クラスである。indexing_suite が要求するポリシーをすべて提供する。

使用例：

```
class X {...};
...

class_<std::vector<X> >("XVec")
    .def(vector_indexing_suite<std::vector<X> >())
;
```

XVec は完全な Python コンテナとなる（完全な例と Python のテストも見よ）。

map_indexing_suite［ヘッダ<boost/python/indexing/map_indexing_suite.hpp>］

map_indexing_suite クラスは、std::map クラス（および std::map スタイルのクラス（例：std::map のインターフェイスを持つクラス））をラップするために設計された定義済みの indexing_suite 派生クラスである。indexing_suite が要求するポリシーをすべて提供する。

使用例：

```
class X {...};
...

class_<std::map<X> >("XMap")
    .def(map_indexing_suite<std::map<X> >())
;
```

既定では添字アクセスした要素はプロキシで返される。NoProxy テンプレート引数で true を与えるとこれは無効化できる。XMap は完全な Python コンテナとなる（完全な例と Python のテストも見よ）。

indexing_suite クラス

indexing_suite<class Container, class DerivedPolicies, bool NoProxy, bool NoSlice, class Data, class Index, class Key>

テンプレート引数	要件	セマンティクス	既定
Container	クラス型	Python に対してラップするコンテナ型。	
DerivedPolicies	`indexing_suite` の派生クラス	ポリシーフックを提供する派生クラス群。以下の DerivedPolicies を見よ。	
NoProxy	論理値	既定では添字アクセスした要素は Python の参照のセマンティクスを持ち、プロキシにより返される。これは *NoProxy* テンプレート引数に**真**を与えることで無効化できる。	`false`
NoSlice	論理値	スライスを許可しない。	`false`
Data		コンテナのデータ型。	`Container::value_type`
Index		コンテナの添字型。	`Container::size_type`
Key		コンテナのキー型。	`Container::value_type`

```
template <
      class Container
    , class DerivedPolicies
    , bool NoProxy = false
    , bool NoSlice = false
    , class Data = typename Container::value_type
    , class Index = typename Container::size_type
    , class Key = typename Container::value_type
>
class indexing_suite
    : unspecified
{
public:

    indexing_suite(); // デフォルトコンストラクタ
}
```

DerivedPolicies

派生クラスは `indexing_suite` が必要なフックを提供する。

```
data_type&
get_item(Container& container, index_type i);

static object
get_slice(Container& container, index_type from, index_type to);

static void
set_item(Container& container, index_type i, data_type const& v);
```

```
static void
set_slice(
    Container& container, index_type from,
    index_type to, data_type const& v
);

template <class Iter>
static void
set_slice(Container& container, index_type from,
    index_type to, Iter first, Iter last
);

static void
delete_item(Container& container, index_type i);

static void
delete_slice(Container& container, index_type from, index_type to);

static size_t
size(Container& container);

template <class T>
static bool
contains(Container& container, T const& val);

static index_type
convert_index(Container& container, PyObject* i);

static index_type
adjust_index(index_type current, index_type from,
    index_type to, size_type len
);
```

これらのポリシーの大部分は自己説明的であるが、convert_indexとadjust_indexは少し説明が必要である。

convert_indexはPythonの添字をコンテナが処理可能なC++の添字に変換する。例えばPythonにおける負の添字は右から数え始める（例：C[-1]はC内の最も右の要素を差す）。convert_indexはC++コンテナのために必要な変換を処理しなければならない（例：-1はC.size()-1である）。convert_indexはまた、添字の型（Pythonの動的型）をC++コンテナが想定する実際の型に変換できなければならない。

コンテナが拡張か縮小すると、要素への添字はデータの移動に追従して調整しなければならない。例えば 5 要素から成るベクタの 0 番目（a）から 3 つの要素を削除すると、添字 4 は添字 1 となる。

```
[a][b][c][d][e] ---> [d][e]
             ^           ^
             4           1
```

adjust_indexの仕事は調整である。添字 *current* を与えると、この関数はコンテナにおける添字 *from..to* におけるデータを *len* 個の要素で置換したときの調整後の添字を返す。

vector_indexing_suite クラス

vector_indexing_suite<class Container, bool NoProxy, class DerivedPolicies>クラステンプレート

テンプレート引数	要件	セマンティクス	既定
Container	クラス型	Python に対してラップするコンテナ型	
NoProxy	論理値	既定では添字アクセスした要素は Python の参照のセマンティクスを持ち、プロキシにより返される。これは *NoProxy* テンプレート引数に**真**を与えることで無効化できる。	`false`
DerivedPolicies	`indexing_suite` の派生クラス	`vector_indexing_suite` はさらに定義済みのポリシーに派生している可能性がある。CRTP（James Coplien の「奇妙に再帰したテンプレートパターン」、C++レポート、1995 年 2 月）を介した静的な多態により基底クラス `indexing_suite` が最派生クラスのポリシー関数を呼び出せる。	

```
template <
    class Container,
    bool NoProxy = false,
    class DerivedPolicies = unspecified_default
class vector_indexing_suite : unspecified_base
{
public:

    typedef typename Container::value_type data_type;
    typedef typename Container::value_type key_type;
    typedef typename Container::size_type index_type;
    typedef typename Container::size_type size_type;
    typedef typename Container::difference_type difference_type;

    data_type&
    get_item(Container& container, index_type i);

    static object
    get_slice(Container& container, index_type from, index_type to);

    static void
    set_item(Container& container, index_type i, data_type const& v);

    static void
    set_slice(Container& container, index_type from,
        index_type to, data_type const& v);

    template <class Iter>
    static void
    set_slice(Container& container, index_type from,
        index_type to, Iter first, Iter last);

    static void
    delete_item(Container& container, index_type i);

    static void
    delete_slice(Container& container, index_type from, index_type to);
```

```
    static size_t
    size(Container& container);

    static bool
    contains(Container& container, key_type const& key);

    static index_type
    convert_index(Container& container, PyObject* i);

    static index_type
    adjust_index(index_type current, index_type from,
        index_type to, size_type len);
};
```

map_indexing_suite クラス

map_indexing_suite<class Container, bool NoProxy, class DerivedPolicies>クラステンプレート

テンプレート引数	要件	セマンティクス	既定
Container	クラス型	Python に対してラップするコンテナ型。	
NoProxy	論理値	既定では添字アクセスした要素は Python の参照のセマンティクスを持ち、プロキシにより返される。これは *NoProxy* テンプレート引数に**真**を与えることで無効化できる。	`false`
DerivedPolicies	`indexing_suite` の派生クラス	`map_indexing_suite` はさらに定義済みのポリシーに派生している可能性がある。CRTP（James Coplien の「奇妙に再帰したテンプレートパターン」、C++レポート、1995 年 2 月）を介した静的な多態により基底クラス `indexing_suite` が最派生クラスのポリシー関数を呼び出せる。	

```
template <
    class Container,
    bool NoProxy = false,
    class DerivedPolicies = unspecified_default
class map_indexing_suite : unspecified_base
{
public:

    typedef typename Container::value_type value_type;
    typedef typename Container::value_type::second_type data_type;
    typedef typename Container::key_type key_type;
    typedef typename Container::key_type index_type;
    typedef typename Container::size_type size_type;
    typedef typename Container::difference_type difference_type;

    static data_type&
    get_item(Container& container, index_type i);
```

```
    static void
    set_item(Container& container, index_type i, data_type const& v);

    static void
    delete_item(Container& container, index_type i);

    static size_t
    size(Container& container);

    static bool
    contains(Container& container, key_type const& key);

    static bool
    compare_index(Container& container, index_type a, index_type b);

    static index_type
    convert_index(Container& container, PyObject* i);
};
```

設定に関する情報

はじめに

Boost.Python は<boost/config.hpp>にある数個の設定マクロのほか、アプリケーションが与える設定マクロを使用する。これらのマクロについて記載する。

アプリケーション定義のマクロ

これらは Boost.Python を使用するアプリケーションが定義可能なマクロである。動的ライブラリをカバーするのに C++標準の厳密な解釈を拡大するのであれば、異なるライブラリ（拡張モジュールや Boost.Python 自身も含む）をコンパイルするときにこれらのマクロの異なる値を使用することは ODR 違反であることに注意していただきたい。しかしながら、この種の違反を検出可能か問題となる C++実装は無いようである。

マクロ	既定	意味
BOOST_PYTHON_MAX_ARITY	15	引数 x1, x2, ...x*n* をとるよう指定した Boost.Python 関数の起動における、ラップする関数、メンバ関数、コンストラクタの最大引数長。これには特に object::operator()(...) や call_method<R>(...) のようなコールバック機構も含まれる。
BOOST_PYTHON_MAX_BASES	10	ラップした C++クラスの基底型を指定する bases<...>クラステンプレートのテンプレート引数の最大数。
BOOST_PYTHON_STATIC_MODULE	（未定義）	定義すると、モジュール初期化関数がエクスポート対象シンボルとして扱われないようになる（コード内での区別をサポートするプラットフォームの場合）。
BOOST_PYTHON_ENABLE_CDECL	（未定義）	定義すると、__cdecl 呼び出し規約を使用する関数のラップが可能となる。
BOOST_PYTHON_ENABLE_STDCALL	（未定義）	定義すると、__stdcall 呼び出し規約を使用する関数のラップが可能となる。
BOOST_PYTHON_ENABLE_FASTCALL	（未定義）	定義すると、__fastcall 呼び出し規約を使用する関数のラップが可能となる。

ライブラリ定義の実装マクロ

これらのマクロは Boost.Python が定義するものであり、新しいプラットフォームへ移植する実装者のみが取り扱う実装の詳細である。

マクロ	既定	意味
BOOST_PYTHON_TYPE_ID_NAME	（未定義）	定義すると、共有ライブラリ境界をまたいだ type_info の比較がこのプラットフォームでは動作しないことを指定する。言い換えると、shared-lib-2 内の typeid(T) を比較する関数に shared-lib-1 が typeid(T) を渡すと、比較結果は false になるということである。このマクロを定義しておくと、Boost.Python は std::type_info オブジェクトの比較を直接使用する代わりに typeid(T).name() の比較を使用する。

マクロ	既定	意味
BOOST_PYTHON_NO_PY_SIGNATURES	（未定義）	定義すると、モジュール関数のドキュメンテーション文字列に対して Python のシグニチャが生成されなくなり、モジュールが登録した変換器に Python 型が紐付かなくなる。また、モジュールのバイナリサイズが約 14%（gcc でコンパイルした場合）削減する。boost_python 実行時ライブラリで定義すると、docstring_options.enable_py_signatures() の既定は false に設定される。
BOOST_PYTHON_SUPPORTS_PY_SIGNATURES	BOOST_PYTHON_NO_PY_SIGNATURES を定義していないと定義される	このマクロを定義すると、Python のシグニチャをサポートしない古いバージョンの Boost.Python からのスムースな移行が有効になる。使用例はここを見よ。
BOOST_PYTHON_PY_SIGNATURES_PROPER_INIT_SELF_TYPE	（未定義）	定義すると、__init__メソッドの self 引数の Python 型を適切に生成する。それ以外の場合、object を使用する。モジュールのバイナリサイズが約 14%（gcc でコンパイルした場合）増加するため、既定では定義されない。

動作を確認したプラットフォームとコンパイラ

最新の情報は退行ログを確認いただきたい。他でマークされていないログはリリースの状態ではなく、CVS の状態を反映する。

以前のバージョンの Boost.Python は以下のプラットフォームとコンパイラでテストに成功した。

Unix プラットフォーム

Python 2.2 および 2.2.2b1

- RedHat Linux 7.3（Intel x86）上の GCC 2.95.3、2.96、3.0.4、3.1 および 3.2
- OSF v. 5.1（Dec/Compaq Alpha）上の Tru64 CXX 6.5.1
- IRIX 6.5（SGI mips）上の MIPSPro 7.3.1.2m
- SunOS 5.8 上の GCC 3.1

Python 2.2.1

- OSF v. 5.1（Dec/Compaq Alpha）上の KCC 3.4d
- AIX 上の KCC 3.4d

Microsoft Windows XP Professional

Python 2.2、2.2.1 および 2.2.2b1

- Microsoft Visual C++ 6、7 および 7.1 ベータ
- Microsoft Visual C++ 6 で STLPort 4.5.3 を使用
- Metrowerks CodeWarrior 7.2、8.0、8.2 および 8.3 ベータ
- Intel C++ 5.0、6.0 および 7.0 ベータ
- Intel C++ 5.0 で STLPort 4.5.3
- Cygwin GCC 3.0.4 および 3.2
- MinGW-1.1（GCC 2.95.3-5）
- MinGW-2.0（GCC 3.2）

定義

arity

関数、メンバ関数が受け取る引数の数。特に指定がない限り、メンバ関数の不可視な `this` 引数は数に含まれない。

ntbs

null 終了バイト文字列（Null-Terminated Byte String）、または'C'文字列。C++の文字列リテラルは ntbs である。ntbs は null であってはならない。

raise

Python において例外は、C++のように「投げられる（thrown）」のではなく「送出される（raised）」。このドキュメントにおいて C++コードの文脈で Python 例外が「送出される」とは、対応する Python 例外が Python/'C' API で設定され `throw_error_already_set()` が呼び出されるという意味である。

POD

C++標準における技術用語で、「古き良き単純なデータ（Plain Ol'Data）」の短縮形。POD 構造体は、非静的データメンバへのポインタ型、POD 構造体、POD 共用体（またはそれらの型の配列）か参照のいずれも持たず、ユーザ定義コピー代入演算子もユーザ定義デストラクタも持たない集約クラスである。同様に POD 共用体は、非静的データメンバへのポインタ型、POD 構造体、POD 共用体（またはそれらの型の配列）か参照のいずれも持たず、ユーザ定義コピー代入演算子もユーザ定義デストラクタも持たない集約共用体である。POD クラスは POD 構造体か POD 共用体のいずれかであるクラスである。集約は配列か、ユーザ宣言コンストラクタを持たず（12.1 節）、非公開・限定公開な非静的データメンバを持たず（11 節）、基底クラスを持たず（10 節）、仮想関数を持たない（10.3 節）クラスである（9 節）。

ODR

「定義は 1 つ規則（One Definition Rule）」。C++プログラムにおけるあらゆる実体は、プログラムを構成するすべての翻訳単位（オブジェクトファイル）で同じ定義を持たなければならないということ。

Boost.Python を使用しているプロジェクト

はじめに

これは Boost.Python を使用しているプロジェクトの部分的なリストである。あなたが Python/C++バインディングに Boost.Python を使用しているのであれば、このページにあなたのプロジェクトを追加をさせていただきたい。プロジェクトの短い説明と Boost.Python を使用して解決した問題について投稿してもらえれば、このページに追加しよう。

データ解析

NeuraLab

NeuraLab は、Neuralynx acquisition systems による神経データに特化したデータ解析環境である。Neuralab はプレゼンテーション水準のグラフィクス、数値解析ライブラリおよび Python スクリプトエンジンを 1 つのアプリケーションにまとめたものである。Neuralab を使用すると、Neuralynx のユーザはマウスを数回クリックするだけで一般的な解析ができる。上級ユーザはカスタムの Python スクリプトを作成でき、必要があればメニューに割り当てられる。

TSLib – Fortress Investment Group LLC

Fortress Investment Group は、C++による内部財務解析ツールの開発と Boost.Python によるそれらの Python バインディングの整備のために Boost Consulting と契約した。

Fortress の Tom Barket はこう書いている:

> *私たちには財務と経済の調査に特化した巨大な C++の解析ライブラリがある。速度と業務上枢要な安定性を主眼に構築した。一方で Python は、新しいアイデアをすばやく試してみたり、C++の場合より生産性を向上させる柔軟性を与えてくれる。Python を際立たせる重要な機能がいくつもある。その優雅さ、安定性、Web における資源の幅広さはすべて貴重なものであるが、最も重要なものはオープンソースの透明性がもたらす拡張性である。Boost.Python は Python の多大なパワーと制御を維持しながら、同時にその拡張性を極限まで単純で直感的にする。*

教育

Kig

KDE Interactive Geometry は、KDE デスクトップ向けに構築されたハイスクールレベルの教育ツールである。学生が幾何学的な構造を扱うのによいツールである。この種のアプリケーションで最も直感的かつ機能が豊富なものといえる。

0.6.x 以降のバージョンではユーザが Python 言語内で構築したオブジェクトをサポートする（予定である）。関連する内部 API のエクスポートに Boost.Python を使用することで、それらの処理が非常に簡単になっている。

エンタープライズ

OpenWBEM

OpenWBEM プロジェクトは商用・非商用アプリケーション向けの Web Based Enterprise Management のオープンソース実装を開発している。

Dan Nuffer はこう書いている：

> *OpenWBEM のクライアント API をラップするのに Boost.Python を使っている。WBEM を使用する管理ソリューションを開発するときに、高速なプロトタイピング、テスト、スクリプティングが可能になった。*

Metafaq

Transversal, Inc. による Metafaq は、企業レベルのオンラインナレッジベース管理システムである。

Ben Young はこう書いている：

> *複数のバックエンドとフロントエンドを介してエクスポートする API に対して、Python バインディングを生成する自動処理に Boost.Python を使用している。おかげで完全なコンパイルサイクルに入ることなく、簡単なテストや一度限りのスクリプトを書けるようになった。*

ゲーム

Civilization IV

「Sid Meier の Civilization IV は売り上げ 5 百万部以上の PC 戦略シリーズの 4 作目です。壮観な新規 3D グラフィクスと完全に新しい 1 人・多人数コンテンツを備え、フランチャイズに向け大きく前進しました。また Civilization IV ではプレイヤー自身が Python や XML でアドオンを作成でき、ユーザ変更の新基準を打ち立てます。

Sid Meier の Civilization IV は 2005 年の暮れ、PC 向けにリリース予定です。http://www.firaxis.com にお越しいただくか、kgilmore@firaxis.com までお問い合わせください。」

> *C++のゲームコードと Python 間のインターフェイス層で Boost.Python を使用している。マップの生成、インターフェイススクリーン、ゲームのイベント、ツール、チュートリアル等の多くの目的で Python を使用している。ゲームをカスタマイズする必要のある MOD 製作者に対して、最高水準のゲーム操作をエクスポートしている。*
>
> *- Mustafa Thamer (Civ4 メインプログラマ)*

Vega Strike

Vega Strike は、広大な宇宙空間でトレードや賞金稼ぎをする 3 次元空間シミュレータである。プレイヤーは海賊や異星人に遭遇し、危険に直面しながら様々な決定を下していく。

Vega Strike ではスクリプト機能に Python を使うことに決め、Python のクラス階層と C++のクラス階層の橋渡しに Boost を使用した。その結果、ミッションの設計と AI の記述時に、ユニットをネイティブな Python のクラスとして扱う非常に柔軟なスクリプトシステムが完成した。

現在、巨大な経済・惑星シミュレーションが現在 Python 内のバックグランドで走り、その結果はプレイヤーが近くにいる場合に Python 内で接近してシミュレートされた世界の近くで現れる様々なグループの宇宙船の形で C++に戻される。

グラフィクス

OpenSceneGraph Bindings

Gideon May は、クロスプラットフォームの C++/OpenGL リアルタイム可視化ライブラリである OpenSceneGraph に対するバインディングを作成した。

HippoDraw

HippoDraw は、ヒストグラムや散布図等を描画するキャンバスで構成されたデータ解析環境である。高度に対話的な GUI を有するが、場合によってはスクリプトを用意しなければならない。HippoDraw は Python の拡張モジュールとして動作させることで、あらゆる操作を Python と GUI の両方から実行できる。

Web ページがオンラインになる以前、Paul F. Kunz はこう書いている:

> *プロジェクトのために Web ページを用意すべきではないが、組織のページは http://www.slac.stanford.edu（アメリカ最初の Web サーバサイト）である。*

とても面白い雑学だったので載せることにした。

IPLT

Ansgar Philippsen はこう書いている:

> *IPLT はイメージ処理ライブラリであり、構造生物学・電子顕微鏡コミュニティのツールボックスである。今のところ製品化の段階ではないが活発に開発が進んでおり、私の中では新進気鋭のプロジェクトである。Python はメインのスクリプティング・対話水準で使用しているが、背後の C++クラスライブラリが Boost.Python により（少なくとも高水準インターフェイスは）完全にエクスポートされているため、高速なプロトタイピングにも使用している。このプロジェクトにおける C++と Python の組み合わせは素晴らしいとしか言いようがない。*

PythonMagick

PythonMagick は、GraphicsMagick イメージ操作ライブラリの Python に対するバインディングである。

VPython

Bruce Sherwood はこう書いている:

> *VPython は操縦可能な 3D アニメーションを簡単に作成できる Python 拡張である。計算コードの副作用として生成される。VPython は物理やプログラミングの授業といった教育目的に使用されているが、博士研究員がシステムやデータを 3D で可視化するのに使用されたこともある。*

科学計算

CAMFR

CAMFR は光通信学、電磁気学のモデリングツールである。計算操作に Python を使用している。

Peter Bienstman はこう書いている:

> *素晴らしいツールを提供してくれてありがとう！*

cctbx – Computational Crystallography Toolbox

Computational Crystallography は、X 線回折の実験データからの結晶構造の原子モデルの導出を扱う。cctbx は結晶学の計算を行う基本的なアルゴリズムのオープンソースライブラリである。コアアルゴリズムは C++で実装されており、高水準な Python インターフェイスを介してアクセスする。

cctbx は Boost.Python とともに開発が進められ、Python/C++ハイブリッドシステムとしての基盤にもとづいて設計された。1 つの些細な例外を除き、実行時の多態性は Python により完全に処理される。C++のコンパイル時多態性はパフォーマンス重視のアルゴリズムを実装するのに使用されている。Python と C++の両方の層は Boost.Python を使用して統合されている。

SourceForge の cctbx プロジェクトは非結晶学的なアプリケーションでの使用を容易にするモジュール群で構成される。scitbx モジュールは科学アプリケーションのための汎用配列ファミリ、および FFTPACK[22]と L-BFGS[23]準ニュートンミニマイザを実装する。

EMSolve

EMSolve は、電荷保存・エネルギー保存のマクスウェルの方程式のための安定したソルバである。

Gaudi および RootPython

Gaudi は、CERN における LHCb および ATLAS 実験[24]の過程で開発された粒子物理衝突データ処理アプリケーションである。

Pere Mato Vila はこう書いている:

> *当フレームワークにスクリプト可能かつ対話的な機能を与えるために Boost.Python を使用している。Python からあらゆるフレームワークサービスおよびアルゴリズムに対話的にアクセスできるよう、「GaudiPython」というモジュールを Boost.Python を使用して実装した。RootPython もまた、ROOT フレームワークと Python 間の汎用的な「ゲートウェイ」を提供するのに Boost.Python を使用する。*
>
> *Boost.Python は素晴らしい。当フレームワークの Python に対するインターフェイスすばやく構築できた。私たちは物理学者（エンドユーザ）に Python ベースの高速な解析アプリケーション開発環境を容易にするよう試みているところであり、Boost.Python は本質的な役割を果たしている。*

22 訳注 FFTPACK（http://ja.wikipedia.org/wiki/FFTPACK）

23 訳注 Limited-memory Broyden-Fletcher-Goldfarb-Shanno 法（http://en.wikipedia.org/wiki/L-BFGS）

24 訳注 LHC アトラス実験（http://atlas.kek.jp/）

ESSS

ESSS（Engineering Simulation and Scientific Software）は、ブラジルおよび南アメリカのマーケットにおいて工学ソリューションの提供、および計算流体力学と画像解析に関する製品とサービスを提供する活動を行う企業である。

Bruno da Silva de Oliveria はこう書いている：

> *私たちの仕事は、最近 C++による排他的なものから Python と C++を使ったハイブリッド言語のアプローチへ移行した。二者の間の層を与えてくれたのは Boost.Python であり、非常に素晴らしい結果となった。*

このテクノロジーを用いてこれまでに 2 つのプロジェクトを開発してきた。

Simba は、滑油システムの発展のシミュレーションから収集した地層の 3 次元可視化を提供する。これにより、ユーザはシミュレーションの時間に沿った変形、圧力および流体といったシミュレーションにおける様々な側面からの解析が可能である。

Aero の狙いはブラジルの様々な企業や大学の技術を用いた CFD[25]の構築である。ESSS は GUI や結果のポストプロセスといった種々のアプリケーションモジュール群を扱う。

PolyBoRi

Michael Brickenstein はこう書いている：

> *PolyBoRi のコアは C++ライブラリであり、多項環や論理変数のべき集合の部分集合のみならず、論理多項・単項式、指数ベクトルのための高水準データ型を提供する。ユニークなアプローチとして、多項式構造の内部記憶型として二分決定図を使用している。この C++ライブラリの最上部で Python のインターフェイスを提供している。これによりグレブナー基底計算における複雑かつ拡張可能な戦略のみならず、複雑な多項式系の解析も可能になる。Boost.Python のおかげでこのインターフェイスの作成は鮮やかなものとなった。*

Pyrap

Ger van Diepen はこう書いている：

> *Pyrap は電波天文学パッケージ casacore（casacore.googlecode.com）に対する Python インターフェイスである。（LOFAR[26]、ASKAP[27]、eVLA[28] のような電波天体望遠鏡で取得した）データを簡単に numpy 配列で得られ、利用可能な多数の Python パッケージを使って基本的な検査と操作が容易になることが、pyrap が天文学者に親しまれている理由である。*
>
> *Boost.Python を使用することで、様々なデータ型（numpy 配列および numpy 配列の個々の要素も含む）の変換器を非常に簡単に作成できるようになった。完全に再帰的に動作する点も優れている。C++関数から Python へのマッピングは直感的に行うことができた。*

RDKit: Cheminformatics and Machine Learning Software

C++と Python で書かれた化学情報工学と機械学習ソフトウェアのコレクションである。

---

25 訳注 Computational Fluid Dynamics。計算流体力学

26 訳注 LOw Frequency ARray（http://ja.wikipedia.org/wiki/LOFAR）

27 訳注 Australian Square Kilometre Array Pathfinder（http://en.wikipedia.org/wiki/Australian_Square_Kilometre_Array_Pathfinder）

28 訳注 expanded Very Large Array。拡大超大型干渉電波望遠鏡群（http://ja.wikipedia.org/wiki/超大型干渉電波望遠鏡群）

システムライブラリ

Fusion

Fusion は、C++におけるプロトコルの実装をサポートするライブラリである。Twisted[29]とともに使用し、メモリ確保戦略や高速なメソッド内部呼び出し等の制御を提供する。Fusion は TCP、UDP およびマルチキャストをサポートし、Python バインディングに Boost.Python を使用している。

Fusion は MIT ライセンスのもとで http://itamarst.org/software からダウンロードできる。

ツール

Jayacard

Jayacard は、非接触スマートカードのための安全でポータブルなオープンソースオペレーティングシステム、およびスマートカード OS とアプリケーション開発を容易にする高品質な開発ツール群の完全なセットの開発を主眼に置いている。

スマートカードリーダ管理のコア部分は C++で書かれているが、開発ツールはすべて友好的な Python 言語で書かれている。Boost.Python は、スマートカードリーダのコアライブラリに対するツールのバインディングにおいて根本的な役割を果たしている。

29 訳注 Twisted は Python で書かれたイベント駆動型のネットワーキングエンジン（http://twistedmatrix.com/）

サポートリソース

概要

以下は、Boost.Python における問題や機能リクエストについて利用可能なサポートリソースのリストである。**Boost.Python の開発者に直接メールでサポートを請うのは遠慮いただきたい。**その代わりに以下のリソースを利用いただきたい。開発者はちゃんと見ているよ！

- **Boost Consulting**[30] – すべての Boost ライブラリについての商用サポート、開発、トレーニング、配布を Boost.Python の提供者より。
- **Python C++-sig** メーリングリストは Python/C++の相互運用、特に Boost.Python について議論するフォーラムである。Boost.Python に関する質問はここに投げるとよい。
- Mike Rovner による **Boost.Python の Wiki ページ**は PythonInfo Wiki の一部であり、人々の経験をかき集めたフォーラムとして、またクックブックとして提供されている。

30 訳注 BoostPro 社はなくなりました。Windows 用インストーラは http://github.com/boostpro/installer/ で公開されています。

よくある質問と回答

関数ポインタを引数にとる関数をラップするにはどうすればよいか

次のようにしたい場合は、

```
typedef boost::function<void (string s) > funcptr;

void foo(funcptr fp)
{
    fp("hello,world!");
}

BOOST_PYTHON_MODULE(test)
{
    def("foo",foo) ;
}
```

Python 側はこう。

```
>>> def hello(s):
...     print s
...
>>> foo(hello)
hello, world!
```

短い答えは「できない」だ。これは Boost.Python の制限ではなく C++の制限による。問題は Python の関数は実際はデータであり、データと C++関数をひとまとめにするには関数の静的変数に格納する以外に方法がないということである。この話の問題は、一握りのデータをすべての関数とひとまとめすることしかできず、`foo` へ渡すと決めた Python 関数についてその場で新しい C++関数をコンパイルする方法はないということである。言い換えると、C++関数が常に**同じ** Python 関数を呼び出すのであれば動作するが、おそらくあなたの希望ではないだろう。

ラップする C++コードを変更することに糸目を付けなければ、代わりにそれを `object` へ渡し呼び出すとよい。多重定義された関数呼び出し演算子は、渡した先の `object` の背後にある Python 関数を呼び出す。

この問題についてより多くの考え方に触れるには、このポスト[31]を見よ。

「懸垂参照を返そうとしました」エラーが出る。何が間違っているのか

厄介なクラッシュの発生を防ぐために発生している例外である。大抵は次のようなコードを書いたときに発生する。

```
period const& get_floating_frequency() const
{
  return boost::python::call_method<period const&>(
```

31 訳注　ActiveState サイトへのリンクは移動してしまいました。http://code.activestate.com/lists/python-cplusplus-sig/以下が移動先と思われますが、訳者には個々のメッセージの場所が分かりませんでした。

```
      m_self,"get_floating_frequency");
}
```

次のようなエラーが発生する。

```
ReferenceError: Attempt to return dangling reference to object of type:
class period
```

この場合、call_method が呼び出す Python のメソッドが新しい Python オブジェクトを構築する。その Python が所有し内包する C++オブジェクト（class period のインスタンス）への参照を返そうとしている。呼び出されるメソッドは短命な新しいオブジェクトを戻すため、それへの参照のみが上記の get_floating_frequency() の存続期間で保持される。関数が返ると Python のオブジェクトが破壊されるため、period クラスのインスタンスが破壊され、返った参照は懸垂したままとなる。もはや未定義の振る舞いであり、この参照で何かしようとするとクラッシュする。Boost.Python はこのような状況を実行時に検出し、クラッシュする前に例外を投げる。

return_internal_reference の効率はどうか

質問

> *12 個の double を持つオブジェクトがある。別のクラスのメンバ関数がこのオブジェクトへの const&を返す。返されるオブジェクトを Python で使用するという観点では、得られるのが戻り値のオブジェクトのコピーと参照のどちらであるかは気にしていない。Boost.Python のバージョン 2 で、copy_const_referece か return_internal_reference のどちらを使用するか決めようと思う。生成されるコードのサイズやメモリオーバーヘッド等、どちらかを選択するのに決め手になるものはあるか。*

回答

copy_const_reference はオブジェクトに対しストレージを使用してインスタンスを作成し、そのサイズは base_size + 12 * sizeof(double)である。return_internal_reference はオブジェクトへのポインタに対しストレージを使用してインスタンスを作成し、そのサイズは base_size + sizeof(void*)である。しかしながら、元のオブジェクトの弱参照リストに入る弱い参照オブジェクトと、内部で参照するオブジェクトの寿命を管理するための特別なコールバックオブジェクトも作成する。私の考えはどうかというと、この場合は copy_const_reference がよいと思う。全メモリ使用量と断片化が減少し、トータルサイクルも削減することだろう。

C++コンテナを引数にとる関数をラップするにはどうすればよいか

Ralf W. Grosse-Kunstleve が次のようなノートを残している。

> *1 番目の方法：通常の class_<>ラッパを使用する。*

```
class_<std::vector<double> >("std_vector_double")
  .def(...)
  ...
  ;
```

> *これをテンプレート内に持っていくと、様々な型を同じコードでラップできる。このテクニックは scitbx パッケージ内のファイル scitbx/include/scitbx/array_family/boost_python/flex_wrapper.h で使用している。このファイルを*

> *std::vector<>インスタンスをラップするよう変更するのは容易である。*
>
> *この種のC++/Pythonの束縛は多数（10000以上）の要素を持つコンテナに最も適している。*
>
> *2番目の方法：カスタムのrvalue変換器を使用する。Boost.Pythonの「rvalue変換器」は次のような関数シグニチャにマッチする。*

```
void foo(std::vector<double> const& array); // const参照渡し
void foo(std::vector<double> array); // 値渡し
```

> *ファイルscitbx/include/scitbx/boost_python/container_conversions.hにいくつかrvalue変換器の実装がある。このコードを使えば、std::vector<>やstd::list<>といったC++コンテナ型からPythonのタプルへの変換、あるいはその逆が可能である。ファイルscitbx/array_family/boost_python/regression_test_module.cppに簡単な例がある。*
>
> *自動的なC++コンテナ－Pythonタプルの変換は、中サイズのコンテナに最も適している。これらの変換器が生成するオブジェクトコードは1番目の方法に比較して著しく小さい。*

2番目の方法の欠点は+、-、*、/、%といった算術演算子が利用できないことである。タプルの代わりに「math_array」型へ変換するカスタムのrvalue変換器があると便利だろう。現時点では実装されていないが、数週間以内にリリースするBoost.Python V2のフレームワークで可能になる（2002年3月10日のポスト）。

std::vector<>－Pythonリスト間の「カスタムのlvalue変換器」もあると便利だろう。これらの変換器はC++からのPythonリストの変更をサポートする。例えば、

C++側：

```
void foo(std::vector<double>& array)
{
  for(std::size_t i=0;i<array.size();i++) {
    array[i] *= 2;
  }
}
```

Python側：

```
>>> l = [1, 2, 3]
>>> foo(l)
>>> print l
[2, 4, 6]
```

カスタムのlvalue変換器についてはBoost.Pythonコアライブラリの変更が必要であり、現時点では利用できない。

追伸：上で触れたscitbxファイル群は匿名CVSで利用できる。

```
cvs -d:pserver:anonymous@cvs.cctbx.sourceforge.net:/cvsroot/cctbx login
cvs -d:pserver:anonymous@cvs.cctbx.sourceforge.net:/cvsroot/cctbx co scitbx
```

致命的なエラー C1204:コンパイラの制限：内部構造がオーバーフローしました。

質問

> *大きなソースファイルをコンパイルすると、このエラーメッセージが出る。どうすればよいか。*

回答

選択肢が 2 つある。

1. コンパイラをアップグレードする（推奨）。
2. ソースファイルを複数の翻訳単位に分割する。

my_module.cpp:

```
...
void more_of_my_module();
BOOST_PYTHON_MODULE(my_module)
{
   def("foo", foo);
   def("bar", bar);
   ...
   more_of_my_module();
}
```

more_of_my_module.cpp:

```
void more_of_my_module()
{
   def("baz", baz);
   ...
}
```

class_<...>宣言を単一のソースファイルに押し込むことがエラーにより不可能な場合、class_オブジェクトへの参照を他のソースファイルの関数へ渡して、その補助ソースファイル内でメンバ関数（.def(...)等）を呼び出すとよい。

more_of_my_class.cpp:

```
void more_of_my_class(class_<my_class>& x)
{
   x
     .def("baz", baz)
     .add_property("xx", &my_class::get_xx, &my_class::set_xx)
     ;

   ...
}
```

Python 拡張をデバッグするにはどうすればよいか

Greg Burley が Unix GCC ユーザに対して以下の回答をしている。

> *C++ライブラリかクラスについてBoost.Python 拡張を作成すると、コードのデバッグが必要になる。結局のところ、Python でライブラリをラップする理由の 1 つがこれだ。BPL を使用することで期待される副作用や利益は、Python のコードが最小限の状況で boost::python が動作しない場合（すなわち、ラップするメソッドが正しくないとエラーが出るが、そのほとんどはコンパイラが捕捉するだろう）でも、デバッグがテスト段階の C++ライブラリに隔離できるということである。*
>
> *gdb セッションを始めて Python による C++ライブラリのデバッグを行うための基本的なステップを以下に示す。あなたの BPL モジュール my_ext.so を含むディレクトリで gdb セッションを開始しなければならないことに注意していただきたい。*

```
(gdb) target exec python
(gdb) run
 >>> from my_ext import *
 >>> [C-c]
(gdb) break MyClass::MyBuggyFunction
(gdb) cont
 >>> pyobj = MyClass()
 >>> pyobj.MyBuggyFunction()
Breakpoint 1, MyClass::MyBuggyFunction ...
Current language:  auto; currently c++
(gdb) do debugging stuff
```

Greg の方法はステップ実行したソースファイルの各行が表示されるので、Emacs の gdb コマンドより優れたものである。

Windows における私のお気に入りのデバッグツールは Microsoft Visual C++ 7 に付属のデバッガだ。このデバッガは、Microsoft および Metrowerks ツールセットのすべてのバージョンが生成するコードで動作するようである。安定していて、ユーザが特別なトリックを使わなくても「とりあえず動作する」。

Raoul Gough は Windows 上の gdb について以下を提供している。

> *最近 gdb の Windows DLL サポートが改善され、少しのトリックで Python 拡張をデバッグできるようになった。まず、DLL から最小限のシンボルを抽出する機能をサポートした最新の gdb が必要である。バージョン 6 以降の gdb か Cygwin gdb-20030214-1 以降が対応している。適切なリリースであれば gdb.info ファイルに Configuration – Native – Cygwin Native – Non-debug DLL symbols 節がある。本稿で概略を示す方法について、この info 節に詳細がある。*
>
> *次に、^C で実行を中断するのではなく Python インタープリタ内にブレークポイントを設定する必要がある。ブレークポイントを設定する適切な場所は `PyOS_Readline` である。Python の対話コマンドを読み込む直前に毎回実行が停止する。デバッガが開始したらブレークポイントを設定可能になる前に Python を開始して自身の DLL を読み込まなければならない。*

```
$ gdb python
GNU gdb 2003-09-02-cvs (cygwin-special)
[...]

(gdb) run
Starting program: /cygdrive/c/Python22/python.exe
Python 2.2.2 (#37, Oct 14 2002, 17:02:34) [MSC 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> ^Z


Program exited normally.
(gdb) break *&PyOS_Readline
Breakpoint 1 at 0x1e04eff0
```

```
(gdb) run
Starting program: /cygdrive/c/Python22/python.exe
Python 2.2.2 (#37, Oct 14 2002, 17:02:34) [MSC 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.

Breakpoint 1, 0x1e04eff0 in python22!PyOS_Readline ()
   from /cygdrive/c/WINNT/system32/python22.dll
(gdb) cont
Continuing.
>>> from my_ext import *

Breakpoint 1, 0x1e04eff0 in python22!PyOS_Readline ()
   from /cygdrive/c/WINNT/system32/python22.dll
(gdb) # my_ext now loaded (with any debugging symbols it contains)
```

Boost.Build で拡張をデバッグする

Boost.Build で boost-python-runtest 規則を使用して拡張モジュールのテストを起動する場合、bjam コマンドラインに --debugger=*debugger* を追加して好きなデバッガを起動できる。

```
bjam -sTOOLS=vc7.1 "--debugger=devenv /debugexe" test
bjam -sTOOLS=gcc -sPYTHON_LAUNCH=gdb test
```

テストを走らせるときに-d+2 オプションを追加すると、Boost.Build がテストを起動するのに使用する完全なコマンドを表示するので非常に便利である。このためには `PYTHONPATH` およびデバッガが正しく動作するのに必要な `LD_LIBRARY_PATH` のような他の重要な環境関数がセットアップされていなければならない。

私の*=演算子が動作しないのはなぜか

質問

> *多数の多重定義演算子とともにクラスを Python へエクスポートした。他はちゃんと動作するのに、*=演算子だけが正しく動作しない。毎回「シーケンスは非 int 型と乗算できません」と言われる。`p1 *= p2` の代わりに `p1.__imul__(p2)` とすると、コードの実行は成功する。私の何が間違っているのか。*

回答

あなたは何も間違っていない。これは Python 2.2 のバグだ。Python 単体でも同じことが起こるはずである（Python 単体で新形式のクラスを使ってみると、Boost.Python 内で何が起こっているか理解できるだろう）。

```
>>> class X(object):
...     def __imul__(self, x):
...         print 'imul'
...
>>> x = X()
>>> x *= 1
```

この問題を解決するには Python をバージョン 2.2.1 以降へアップグレードする必要があり、他の方法はない。

Boost.Python は Mac OS X で動作するか

10.2.8 および 10.3 では Apple の gcc 3.3 コンパイラで動作することが分かっている。

```
gcc (GCC) 3.3 20030304 (Apple Computer, Inc. build 1493)
```

10.2.8 の場合は gcc の 2003 年 8 月アップデートを入手する（http://connect.apple.com/で無償配布されている）。10.3 の場合は Xcode Tools バージョン 1.0 を入手する（こちらも無償である）。

Python 2.3 が必要である。10.3 に付属の Python がよい。10.2.8 では次のコマンドを使用して Python をフレームワークとしてインストールする。

```
./configure -enable-framework
make
make frameworkinstall
```

ターゲットディレクトリが/Library/Frameworks/Python.framework/Versions/2.3であるので、最後のコマンドは root 権限が必要である。しかしながら、このインストールは 10.2.8 に付属の Python バージョンと競合しない。

コンパイルの前に stacksize を増やしておくことも肝要である。例えば次のようにする。

```
limit stacksize 8192k
```

stacksize が小さいと内部コンパイラエラーが出てビルドがクラッシュする場合がある。

boost::python::class_<your_type>テンプレートの実体化をコンパイル中に、たまに Apple のコンパイラが以下のようなエラーを印字（バグ）することがある。

```
.../inheritance.hpp:44: error: cannot
 dynamic_cast `p' (of type `struct cctbx::boost_python::<unnamed>::add_pair*
   ') to type `void*' (source type is not polymorphic)
```

一般的な回避方法はないが、your_type の定義を以下のように修正するとすべての場合で動作するようだ。

```
struct your_type
{
  // メンバデータを定義する前
#if defined(__MACH__) && defined(__APPLE_CC__) && __APPLE_CC__ == 1493
  bool dummy_;
#endif
  // 例えばここにメンバデータを置く
  double x;
  int j;
  // 以下続く
};
```

C++オブジェクトを保持する既存の PyObject を探し出すにはどうすればよいか

> *「常に保持済みの C++オブジェクトへのポインタを返す関数をラップしたい。」*

方法の 1 つとしては、仮想関数を持つクラスをラップするのに使用する機構をハイジャックすることである。コンストラクタで第 1 引数として `PyObject*`を取り、その `PyObject*`を `self` として格納するラッパクラスを作成する場合、薄いラッパ関数内でラッパ型へダウンキャストして元に戻すことができる。例えば、

```
class X { X(int); virtual ~X(); ... };
X* f();  // Python オブジェクトが管理する X を返す


// ラップのためのコード

struct X_wrap : X
{
    X_wrap(PyObject* self, int v) : self(self), X(v) {}
    PyObject* self;
};

handle<> f_wrap()
{
    X_wrap* xw = dynamic_cast<X_wrap*>(f());
    assert(xw != 0);
    return handle<>(borrowed(xw->self));
}

...

def("f", f_wrap());
class_<X,X_wrap,boost::noncopyable>("X", init<int>())
   ...
   ;
```

当然、X が仮想関数を持たない場合、`dynamic_cast` の代わりに実行時チェックを行わない（行わなくてもよい）`static_cast` を使用しなければならない。C++から構築した `x` が `X_wrap` オブジェクトとなることは当然ないため、この方法が動作するのは `x` オブジェクトが Python から構築された場合だけである。

別の方法では C++コードをわずかに変更しなければならない（可能であればこちらのほうがよい）。`shared_ptr<X>`が Python から変換されると、`shared_ptr` は実際は内包する Python オブジェクトへの参照を管理する。逆に `shared_ptr<X>`を Python へ変換すると、ライブラリはそれが「Python オブジェクト管理者」の 1 つであるかチェックし、そうであれば元の Python オブジェクトをそのまま返す。よって `object(p)`と書くだけで Python オブジェクトを戻すことができる。これを利用するには、ラップする C++コードを生のポインタではなく `shared_ptr` で扱えるよう変更可能にしなければならない。

さらに別の方法もある。返したい Python オブジェクトを受け取る関数は、オブジェクトのアドレスと内包する Python オブジェクトの対応関係を記録する薄いラッパでラップでき、このマッピングから Python オブジェクトを捜索する `f_wrap` 関数を用意しておくことができる。

生のポインタの所有権を持つ必要がある関数をラップするにはどうすればいいか

> *私がラップしているAPIの一部分は次のようなものである。*

```
struct A {}; struct B { void add( A* ); }
B::add()は渡されたポインタの所有権を獲得する。
```

> *しかしながら、*

```
a = mod.A()
b = mod.B()
b.add( a )
del a
del b
# メモリの改変により
# Pythonインタープリタがクラッシュする。
```

> *with_custodian_and_wardを使ってaの寿命をbに束縛したとしても、結局のところポインタ先のPythonオブジェクトaが削除されるのを防ぐことはできない。ラップしたC++オブジェクトの「所有権を移動する」方法はあるか。*
>
> — *Bruce Lowery*

ある。C++オブジェクトが auto_ptr に保持されるようにしておく。

```
class_<A, std::auto_ptr<A> >("A")
    ...
    ;
```

次に auto_ptr 引数をとる薄いラッパ関数を作成する。

```
void b_insert(B& b, std::auto_ptr<A> a)
{
    b.insert(a.get());
    a.release();
}
```

これを B.add でラップする。manage_new_object が返すポインタもまた auto_ptr で保持されているため、この所有権の移動が正しく動作することに注意していただきたい。

コンパイルに時間がかかりメモリも大量に消費する！高速化するにはどうすればよいか

チュートリアルの「コンパイルにかかる時間を短縮する」の節を参照いただきたい。

Boost.Python を使用してサブパッケージを作成するにはどうすればよいか

チュートリアルの「パッケージを作成する」の節を参照いただきたい。

error C2064：2 引数を取り込む関数には評価されません

Niall Douglas が次のノートを提供している。

> *Microsoft Visual C++ 7.1 (MS Visual Studio .NET 2003）で以下のようなエラーメッセージが出る場合、ほとんどはコンパイラのバグである。*

```
boost\boost\python\detail\invoke.hpp(76):
error C2064: 2 引数を取り込む関数には評価されません"
```

> *このメッセージは以下のようなコードで引き起こされる。*

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

using namespace boost::python;

class FXThread
{
public:
    bool setAutoDelete(bool doso) throw();
};

void Export_FXThread()
{
    class_< FXThread >("FXThread")
        .def("setAutoDelete", &FXThread::setAutoDelete)
    ;
}
```

> *このバグは `throw()` 修飾子が原因である。回避方法は修飾子を取り除くことである。例えば、*

```
        .def("setAutoDelete", (bool (FXThread::*)(bool)) &FXThread::setAutoDelete)
```

> *（このバグは Microsoft に報告済みである。）*

カスタム文字列型と Python 文字列を自動的に相互変換するにはどうすればよいか

Ralf W. Grosse-Kunstleve が次のノートを提供している。

> *以下は、必要なものがすべて揃った小型の拡張モジュールのデモである。次のは対応する簡単なテストである。*

```
import custom_string
assert custom_string.hello() == "Hello world."
assert custom_string.size("california") == 10
```

> *コードを見れば分かるが、*

- カスタムの `to_python` 変換器（容易）：`custom_string_to_python_str`
- カスタムの lvalue 変換器（より多くのコードが必要）：`custom_string_from_python_str`

> *カスタム変換器は、モジュール初期化関数のトップ近傍のグローバルな Boost.Python レジストリに登録する。一度制御フローが登録コードに渡ると、同じプロセス内でインポートしたあらゆるモジュールで Python 文字列の自動的な相互変換が動作するようになる。*

```
#include <boost/python/module.hpp>
#include <boost/python/def.hpp>
#include <boost/python/to_python_converter.hpp>

namespace sandbox { namespace {

  class custom_string
  {
    public:
      custom_string() {}
      custom_string(std::string const& value) : value_(value) {}
      std::string const& value() const { return value_; }
    private:
      std::string value_;
  };

  struct custom_string_to_python_str
  {
    static PyObject* convert(custom_string const& s)
    {
      return boost::python::incref(boost::python::object(s.value()).ptr());
    }
  };

  struct custom_string_from_python_str
  {
    custom_string_from_python_str()
    {
      boost::python::converter::registry::push_back(
        &convertible,
        &construct,
        boost::python::type_id<custom_string>());
    }

    static void* convertible(PyObject* obj_ptr)
    {
      if (!PyString_Check(obj_ptr)) return 0;
      return obj_ptr;
    }

    static void construct(
      PyObject* obj_ptr,
      boost::python::converter::rvalue_from_python_stage1_data* data)
    {
      const char* value = PyString_AsString(obj_ptr);
      if (value == 0) boost::python::throw_error_already_set();
      void* storage = (
        (boost::python::converter::rvalue_from_python_storage<custom_string>*)
          data)->storage.bytes;
      new (storage) custom_string(value);
      data->convertible = storage;
    }
  };

  custom_string hello() { return custom_string("Hello world."); }
```

```
  std::size_t size(custom_string const& s) { return s.value().size(); }

  void init_module()
  {
    using namespace boost::python;

    boost::python::to_python_converter<
      custom_string,
      custom_string_to_python_str>();

    custom_string_from_python_str();

    def("hello", hello);
    def("size", size);
  }

}} // namespace sandbox::<anonymous>

BOOST_PYTHON_MODULE(custom_string)
{
  sandbox::init_module();
}
```

Python への自動変換器が見つからないのはなぜか

Niall Douglas が次のノートを提供している。

> *上記のようなカスタム変換器を定義すると、メンバデータへの直接アクセスのために `boost::python::class_` が提供する `def_readonly()` および `def_readwrite()` メンバ関数は期待どおりに動作しない。これは `def_readonly("bar", &foo::bar)` が次と等価だからである。*

```
.add_property("bar", make_getter(&foo::bar, return_internal_reference()))
```

> *同様に `def_readwrite("bar", &foo::bar)` は次と等価である。*

```
.add_property("bar", make_getter(&foo::bar, return_internal_reference()),
                     make_setter(&foo::bar, return_internal_reference())
```

> *戻り値のポリシーをカスタム変換に互換性のある形で定義するには、`def_readonly()` および `def_readwrite()` を `add_property()` で置き換える。例えば、*

```
.add_property("bar", make_getter(&foo::bar, return_value_policy<return_by_value>()),
                     make_setter(&foo::bar, return_value_policy<return_by_value>()))
```

インタープリタが複数の場合 Boost.Python はスレッドに対して問題ないか

Niall Douglas が次のノートを提供している。

> *短い答え：ノー。*
>
> *長い答え：解決するパッチは書けるが、困難である。Boost.Python を使用するあらゆるコード（特に仮想関数の多重定義部分）をカスタムのロック・アンロックで囲む必要があり、加えて `boost/python/detail/invoke.hpp` を大幅に修正して Boost.Python があなたのコードを使用するあらゆる部分をカスタムのアンロック・ロックで囲む必要がある。さらに Boost.Python が `invoke.hpp` によりイテレータ変更を起動するときにアンロック・ロック**しない**ように注意しなければならない。*
>
> *パッチを当てた invoke.hpp は C++-SIG メーリングリストにポストされ、アーカイブになっている。機械的に必要な実際の全実装は TnFOX プロジェクト（SourceForge 内の場所）にある。*

Py++ Boost.Python コード生成器

訳注：SourceForge の「C++ Python language bindings」を参照してください。

Pyste Boost.Python コード生成器

訳注：維持されていない機能なので省略します。原文を参照してください。

Boost.Python の裏側

Brett Calcott と David Abrahams のやりとり

> *（訳注：素の文が David Abrahams、引用文が Brett Calcott のメッセージ。以下の 2 問が Brett から David に投げかけられた。その上で）*[32]*これらのケースではいずれもコードを読んでいくことはできるのだが、アーキテクチャの意図がソースからは構造的にも時間的にも分からなかった（つまりその、私が言いたいのはそれらがどのような順序で行われるかだ）。*
>
> *1. 次のようにすると何が起こるか：*

```
struct boring {};
...etc...
class_<boring>("boring")
    ;
```

> *先を続けさせてもらうと、次のようにいろいろ出てくると思うのだが。*

- Python は新しい ClassType の登録を必要とする。
- `boring` 構造体を保持可能な新しい型を構築する必要がある。
- この型に対して内向きと外向きの変換器を登録する必要がある。

> *これらの方法について汎用的な方向性を提示してもらえるだろうか？*

時間の関係で確かこうだったぐらいの回答しかできないけど。このメールを読んだあとにデバッガを使ってコードの詳細を調べるのはどうかな（別に忘れちゃったわけじゃないけどね）。

`Boost.Python.class`（メタ型）を呼び出すと、`Boost.Python.Instance` の新しい（Python の）派生クラスが作成される（`libs/python/src/object/class.cpp` を参照）：

```
>>> boring = Boost.Python.class(
...     'boring'
...   , bases_tuple        # この場合は単に()
...   , {
...         '__module__' : module_name
...       , '__doc__' : doc_string # 省略可能
...     }
... )
```

このオブジェクトに対するハンドルは `registration` の `m_class_object` フィールドで `typeid(boring)` に紐付けされる。（たぶんよろしくないことだけど）拡張モジュールの `boring` 属性を一掃したとしても、レジストリはこのオブジェクトを永久に生存させる。

`class_<boring, non_copyable, ...>`を指定していないので、`boring` の Python への変換器を登録する。この変換器は Python の `boring` オブジェクトが保持する `value_holder` へその引数をコピーする。

32 訳注　元のメッセージは https://mail.python.org/pipermail/cplusplus-sig/2003-July/004480.html。

class<boring, ...>(no_init)を指定していないので、__init__関数オブジェクトをクラスの辞書に追加する。この関数オブジェクトは（ホルダとしてスマートポインタか派生ラッパクラスを指定していないので）Python の boring オブジェクトが保持する value_holder に boring をデフォルトコンストラクトする。

register_class_from_python は、shared_ptr<boring> に対する Python からの変換器を登録するのに使う。boost::shared_ptr はスマートポインタの中でも特殊なものだ。Deleter 引数を使えば、（C++オブジェクトの保持がどのような形態であれ）その内包する C++オブジェクトのみならず Python オブジェクト全体をも管理できるからだ。

bases<>を与えておくと、これら基底クラス群と boring 間の継承図（inheritance.[hpp/cpp]）における関係も登録する。

このコードの以前のバージョンでは、このクラスに対して Python から lvalue への変換器を登録していた。現在はラップされたクラスの Python からの変換は、変換元の Python オブジェクトのメタクラスが Boost.Python のメタクラスである場合、レジストリを調べる前に特殊な場合として処理される。

とまぁ、こういった Python からの変換器は、たぶん明示的に変換を登録しない場合の変換器クラスと同様に扱うべきだね。

> *2. レジストリ内に現れるデータ構造について、手短な概要は*

レジストリは簡単で、typeid から registration への写像（boost/python/converter/registrations.hpp を参照）でしかない。lvalue_chain と rvalue_chain は単に内部的なリンクリストだ。

他に知りたいことがあったら、また聞いてくれ。

継承図について知りたいことがあったら、他のメッセージで個別に聞いてくれ。

> *同時に C++から Python およびその逆方向の型変換について処理の概要はどうか。*

難題だね。背景について調べることを勧めるよ。LLNL 進捗レポート内の関連情報と、そこからリンクしているメッセージを探すといい。あとは、[33]

- http://mail.python.org/pipermail/c++-sig/2002-May/001023.html
- http://mail.python.org/pipermail/c++-sig/2002-December/003115.html
- http://aspn.activestate.com/ASPN/Mail/Message/1280898
- http://mail.python.org/pipermail/c++-sig/2002-July/001755.html

C++から Python への変換:

型と使っている呼び出しポリシーによる。あるいは call<>(...)、call_method<>(...)、object(...)については ref や ptr を使っているかによる。Python への変換は基本的には、「戻り値」の変換（Python から C++の呼び出し）と「引数」の変換（C++から Python の呼び出しと明示的な object()の変換）の 2 つに分けられる。詳細はすぐには思い出せないけど、これら 2 つの振る舞いの違いはとにかくわずかなものだ。上の参考にたぶんその答えがあるし、コードを見たら確実に見つかる。

「既定の」場合だと値による（コピー）変換になるので、Python への変換器として to_python_value を使う。

普通に考えると、ある型を Python へ変換する方法は 1 つしかないはず（スコープ付きのレジストリを使う方法もあるが今は無視しよう）なので、Python への変換は当然テンプレートの特殊化で処理する。この型が組み込みの変換（builtin_converters.hpp）で処理するものであれば、相当する to_python_value のテンプレート特殊化を使う。

33 訳注　Python.org の URL は移動してしまいました。https://mail.python.org/pipermail/cplusplus-sig/以下が移動先と思われますが、訳者には個々のメッセージの場所が分かりませんでした。

上記以外の場合、`to_python_value` は C++型に対する `registration` 内の `m_to_python` 関数を使う。

参照による変換のような他の変換はラップしたクラスでのみ有効で、明示的に要求されるのは `ref(...)` か `ptr(...)` を使うか異なる CallPolicies を指定（異なる Python への変換器を使う）した場合だ。`registration` を使って参照先の C++型に対応する Python クラスを見つける必要があるが、これらの変換器はどこにも登録されない。これらは単に Python の新しいインスタンスを構築し、適当な `Holder` インスタンスを紐付ける。

Python から C++への変換:

もう一度、「戻り値」の変換と「引数」の変換には違いがあることを覚えておこう。そしてその正確な意味は忘れよう。

何が起こるかは lvalue の変換が必要かどうかによる（http://mail.python.org/pipermail/c++-sig/2002-May/001023.html を参照）。rvalue が登録されていれば lvalue は確実に問題ないので、あらゆる lvalue の変換器は型の rvalue の変換チェインにも登録される。

rvalue の変換は、ラップした関数の多重定義と与えられた対象の C++型に対する複数の変換器をサポートするために 2 ステップ必要とする（まず変換が可能か判断して、次のステップで変換したオブジェクトを構築する）。いっぽう、lvalue の変換は 1 ステップで完了できる（オブジェクトへのポインタを得るだけだ。変換が不可能な場合は NULL の可能性がある）。

新着情報・変更履歴

現在の SVN

Python 3 のサポート

現時点の Boost.Python テストはすべてパスした。Boost.Python を使用している拡張モジュールはスムースに Python 3 をサポートすると考えてよい。

`object.contains` を導入した（`x.contains(y)` は Python の `y in x` と等価である）。`dict.has_key` は `object.contains` のラッパに過ぎない。

Python 3 に対してビルドすると、`str.decode` は削除される。

Python 3 に対してビルドすると、`list.sort` の元のシグニチャが次のとおりだったのが

```
void sort(object_cref cmpfunc);
```

次のように変更となる。

```
void sort(args_proxy const &args, kwds_proxy const &kwds);
```

これは Python 3 において `list.sort` がすべての引数がキーワード引数であることを要求するからである。よって呼び出しは次のようにしなければならない。

```
x.sort(*tuple(), **dict(make_tuple(make_tuple("reverse", true))));
```

PEP 3123 に従い、2.6 より前の Python に対して Boost.Python をビルドすると、Boost.Python のヘッダで以下のマクロが定義される。

```
# define Py_TYPE(o)     (((PyObject*)(o))->ob_type)
# define Py_REFCNT(o)   (((PyObject*)(o))->ob_refcnt)
# define Py_SIZE(o)     (((PyVarObject*)(o))->ob_size)
```

よって拡張の作成者はこれらのマクロを直接使用して、コードを簡潔かつ Python 3 と互換にできる。

1.39.0 リリース

- Python のシグニチャが自動的にドキュメンテーション文字列に結合されるようになった。
- ドキュメンテーション文字列の内容を制御するには `docstring_options.hpp` ヘッダを使用せよ。
- この新機能によりモジュールのサイズが約 14% 増加する。これが許容できない場合は、マクロ `BOOST_PYTHON_NO_PY_SIGNATURES` を定義することで無効化できる。このマクロを定義してコンパイルしたモジュールとそうでないモジュールは互換性がある。

- BOOST_PYTHON_NO_PY_SIGNATURES が定義されていない場合、現行のバージョンではマクロ BOOST_PYTHON_SUPPORTS_PY_SIGNATURES が定義される。これにより以前のバージョンの Boost.Python でコンパイルする可能性のあるコードが記述できる（ここを見よ）。
- BOOST_PYTHON_PY_SIGNATURES_PROPER_INIT_SELF_TYPE を定義すると（サイズが 14%増加するが）、__init__メソッドの self 引数に対して適切な Python 型が生成される。
- この新機能をサポートするために to_python_converter.hpp、default_call_policies、ResultConverter、CallPolicies 等に変更が入った。これらはインターフェイスを破壊するような変更にならないようにした。

1.34.0 リリース（2007 年 5 月 12 日）

- C++のシグニチャが自動的にドキュメンテーション文字列に結合されるようになった。
- ドキュメンテーション文字列の内容を制御する docstring_options.hpp ヘッダを新規に追加した。
- 戻り値ポリシーである opaque_pointer_converter による void*と Python の相互変換をサポートした。初期のパッチについて Niall Douglas に感謝する。

1.33.1 リリース（2005 年 10 月 19 日）

- wrapper<T>が *some-smart-pointer*<*T*>の保持型とともに使用できるようになった。
- ビルドで想定する既定の Python のバージョンを 2.2 から 2.4 に変更した。
- Unicode サポートなしでビルドした Python をサポートした。
- アドレス（&）演算子を多重定義したクラスのラップをサポートした。

1.33 リリース（2005 年 8 月 14 日）

- 非静的プロパティのドキュメンテーション文字列をサポートした。
- init<optional<> >および XXX_FUNCTION_OVERLOADS()の最後の多重定義に対してのみクライアントが提供したドキュメンテーション文字列をエクスポートするようにした。
- 組み込み VC++ 4 のサポートをいくつか修正した。
- shared_ptr の Python から rvalue への変換のサポートを強化した。所有する Python オブジェクトが正しい型の NULL の shared_ptr を持たない限り、常に Python オブジェクトを保持するポインタを返す。
- indexing suite を用いた vector<T*>のエクスポートをサポートした。
- MacOS における GCC-3.3 をサポートした。
- Visual Studio のプロジェクトビルドファイルを更新し、新しく 2 つのファイル（slice.cpp および wrapper.cpp）を追加した。
- 索引のページに検索機能を追加した。
- チュートリアルを大幅に修正した。

- MSVC 6 および 7、GCC 2.96、EDG 2.45 のバグ回避コードを大量に追加した。

2005 年 3 月 11 日

間抜けな PyDoc が Boost.Python で動作するようハックを追加した。Nick Rasmussen に感謝する。

1.32 リリース（2004 年 11 月 19 日）

- Boost Software Licese を使用するよう更新した。
- 仮想関数を持つクラスをラップするより優れた方法を新規に実装した。
- 次期 GCC のシンボルエクスポート制御機能のサポートを取り込んだ。Niall Douglas に感謝する。
- `std::auto_ptr` ライクな型のサポートを改良した。
- 関数引数型のトップレベル CV 指定子が関数型の一部分となる Visual C++のバグを回避した。
- 依存関係改善のため、他のライブラリが使用するコンポーネントを `python/detail` 外部、`boost/detail` へ移動した。
- その他のバグ修正とコンパイラバグの回避。

2004 年 9 月 8 日

Python の `Bool` 型をサポートした。Daniel Holth に感謝する。

2003 年 9 月 11 日

- 同じ型に対して複数の to-python 変換器を登録したときに出るエラーを警告に変えた。Boost.Python はメッセージ内に不愉快な型を報告するようになった。
- 組み込みの `std::wstring` 変換を追加した。
- `std::out_of_range` から Python の `IndexError` 例外への変換を追加した。RaoulGough に感謝する。

2003 年 9 月 9 日

`str` に文字の範囲をとる新しいコンストラクタを追加し、ヌル（`'\0'`）文字を含む文字列を受け付けるようになった。

2003 年 9 月 8 日

（`operator()`を持つ）関数オブジェクトからメソッドを作成する機能を追加した。詳細は `make_function` のドキュメントを見よ。

2003 年 8 月 10 日

Roman Yakovenko による新しい `properties` 単体テストを追加し、彼の依頼で `add_static_property` のドキュメントを追加した。

2003 年 8 月 1 日

Nikolay Mladenov による新しい arg クラスを追加した。このクラスは、途中の引数を省略して呼び出せる関数をラップする機能を提供する。

```
void f(int x = 0, double y = 3.14, std::string z = std::string("foo"));

BOOST_PYTHON_MODULE(test)
{
   def("f", f
       , (arg("x", 0), arg("y", 3.14), arg("z", "foo")));
}
```

Python 側は次のようにできる。

```
>>> import test
>>> f(0, z = "bar")
>>> f(z = "bar", y = 0.0)
```

Nikolay に感謝する。

2003 年 7 月 22 日

恐怖のエラー「bad argument type for builtin operation」が出ないようにした。引数エラーで実際の型と想定していた型を表示するようになった。

2003 年 7 月 19 日

Nikolay Mladenov による新しい return_arg ポリシーを追加した。Nikolay に感謝する。

2003 年 3 月 18 日

- Gottfried Ganßauge が不透明ポインタのサポートを提供してくれた。
- Bruno da Silva de Oliveira が素晴らしい Pyste（「Pie-steh」と発音する）パッケージを提供してくれた。

2003 年 2 月 24 日

boost::shared_ptr のサポート強化が完了した。C++クラス x をラップしたオブジェクトが、ラップの方法に関わらず自動的に shared_ptr<X>に変換可能になった。shared_ptr は x オブジェクトだけではなく x を与える Python オブジェクトの寿命を管理し、また逆に shared_ptr を Python に変換するときは元の Python オブジェクトを返す。

2003 年 1 月 19 日

Nikolay Mladenov による `staticmethod` サポートを統合した。Nikolay に感謝する。

2002 年 12 月 29 日

Brett Calcott による Visual Studio のプロジェクトファイルと説明を追加した。Brett に感謝する。

2002 年 12 月 20 日

Python への変換において、多態的なクラス型へのポインタ、参照、スマートポインタの自動的なダウンキャストを追加した。

2002 年 12 月 18 日

各拡張モジュールの各クラスについて個別の変換器を登録する代わりに、共有ライブラリに変換ロジックを配置することにより、from_python 変換を最適化した。

2002 年 12 月 13 日

- `enum_` 値の `scope` 内へのエクスポートが可能になった。
- `signed long` の範囲外の数値を正しく扱うよう、符号無し整数の変換を修正した。

2002 年 11 月 19 日

基底クラスメンバ関数ポインタを `add_property` の引数として使用するときにキャストを不要にした。

2002 年 11 月 14 日

`make_getter` でラップしたクラスデータメンバの自動検出。

2002 年 11 月 13 日

`std::auto_ptr<>`の完全なサポートを追加した。

2002 年 10 月

チュートリアルドキュメントの更新と改良。

2002 年 10 月 10 日

Boost.Python バージョン 2 をリリース！

TODO リスト

クラスのサポート

仮想関数コールバックラッパの基底クラス

- http://aspn.activestate.com/ASPN/Mail/Message/c++-sig/1456023（メッセージの最後を見よ）
- http://mail.python.org/pipermail/c++-sig/2003-August/005297.html（`VirtualDispatcher` で検索するとよい）に、コールバッククラスがその Python ラッパとの関係について所有権を交換する方法が述べられている。
- http://aspn.activestate.com/ASPN/Mail/Message/c++-sig/1860301 に、基底クラスを使ってコールバッククラスを非常に単純化し、「懸垂参照」問題を解決し、オーバーライドされていない仮想関数の呼び出すを最適化する方法が述べられている。

その他

値が重複した enum のサポート

Scott Snyder がパッチを提供している。Dave はある理由から不満だったが、これ以上のアクションが無ければおそらく適用されるだろう（http://aspn.activestate.com/ASPN/Mail/Message/1824616）。

関数

関数オブジェクトのラップ

`operator()` をサポートするクラスは Python のメソッドとしてラップ可能となるだろう（http://mail.python.org/pipermail/c++-sig/2003-August/005184.html）。

多重定義解決の「最良マッチ」

現時点では多重定義の解決は `def` の呼び出し順に依存する（後の多重定義を優先する）。これは常に最良マッチの多重定義を選択するよう変更すべきである。このテクニックはすでに Luabind で培われているので、Langbinding の合流待ちとなっている。

型変換器

非 const な PyTypeObject*から lvalue への変換

http://aspn.activestate.com/ASPN/Mail/Message/C++-sig/1662717

変換器のスコープ制御

- http://article.gmane.org/gmane.comp.python.c++/2044
- 上記が完了すれば Luabind の統合と合流することになるだろう。

boost::tuple

Python との相互変換は問題なさそうである。http://news.gmane.org/find-root.php?message_id=%3cuvewak97m.fsf%40boost%2dconsulting.com%3e を見よ。

FILE*の変換

http://aspn.activestate.com/ASPN/Mail/Message/1411366

void*の変換

CV `void` へのポインタは、不透明な値として関数へ渡したり関数から返したりできるべきである。

呼び出し後（Post-Call）のアクション

Policies オブジェクト内の post-call アクションチェインに from-python 変換器を渡さなければならない（追加のアクションが登録可能な場合）。http://aspn.activestate.com/ASPN/Mail/Message/C++-sig/1755435 の最後を見よ。

PyUnicode のサポート

Lijun Qin によるレビューが http://aspn.activestate.com/ASPN/Mail/Message/C++-sig/1771145 にある。この変更は組み入れる可能性が高い。

所有権のメタデータ

http://aspn.activestate.com/ASPN/Mail/Message/c++-sig/1860301 のスレッドにおいて、Niall Douglas は「偽の」懸垂ポインタ・参照がオブジェクトに関するデータをアタッチすることでエラーを返すという解法のアイデアについて述べた。そのデータの寿命について伝えてこないオブジェクトの参照カウントをフレームワークが決められる。

ドキュメンテーション

組み込みの変換器

組み込みの Python 型と C++型間の組み込みの対応関係についてドキュメントが必要である。

内部的な話

フレームワークの構造についてドキュメントしておく必要がある。Brett Calcott がこのドキュメントをユーザ向けに書き直すと約束し

てくれた。

大規模

スレッドの完全なサポート

Boost.Python におけるスレッドサポートの強化について、多くの人々からパッチが寄せられている（例えば http://aspn.activestate.com/ASPN/Mail/Message/1826544 や http://aspn.activestate.com/ASPN/Mail/Message/1865842 のスレッドを見よ）。唯一の問題はこれらが不完全な解法であることで、完全な解法があるのか検証するには時間と注意が必要である。

Langbinding

このプロジェクトは Boost.Python を一般化して他の言語で動作するもので、一番手は Lua である。http://lists.sourceforge.net/lists/listinfo/boost-langbinding の議論を見よ。

リファクタリングと再構成

http://aspn.activestate.com/ASPN/Mail/Message/c++-sig/1673338

NumArray サポートの強化

http://aspn.activestate.com/ASPN/Mail/Message/C++-sig/1757092 で述べられている強化について統合を検討する。

PyFinalize の安全性

現在のところ、Boost.Python にはグローバル（および関数内静的）オブジェクトが複数あり、それらは Boost.Python 共有オブジェクトがアンロードされるまで参照カウントがゼロにならない。インタープリタが存在しない状態で参照カウントがゼロになるので、クラッシュを引き起こす。`PyFinalize()` の呼び出しを安全にするには、これらのオブジェクトを破壊し Python の参照カウントを解放する `atexit` ルーチンを登録して、Python がインタープリタが存在している間にそれらを始末できるようにしなければならない。Dick Gerrits が何とかすると約束してくれた。

LLNL 進捗レポート

Boost.Python の開発について、Boost Consult と LLNL の規約として 1 か月ごとの進捗報告が求められている。これらのレポートは設計の根拠と関連する議論へのリンクを含む、プロジェクトの有用な記録でもある。

（訳注：本章の翻訳は割愛します。原文を参照してください。）

謝辞

Dave Abrahams は Boost.Python のアーキテクト、設計者、実装者である。

Brett Calcott は Visual Studio プロジェクトファイルとドキュメントに寄与し維持している。

Gottfried Ganßauge は不透明なポインタの変換サポートを提供し、（私が特に依頼しなかったにもかかわらず）ドキュメントと退行テストも付けてくれた！

Joel de Guzman は既定の引数サポートを実装し、素晴らしいチュートリアルを書いてくれた。

Ralf W. Grosse-Kunstleve は pickle サポートを実装し、ライブラリをその誕生から熱狂的にサポートし、設計の決定とユーザの要求に対する非常に貴重な現実世界の見識を提供してくれた。Ralf は、私がすぐにでもライブラリに取り入れたい C++コンテナの変換を行う拡張を書いた。彼はまた、Boost.Python の最初のバージョンでクロスモジュールサポートを実装した。さらに重要なのは、Ralf は大規模ソフトウェアの構築における問題を解決するための C++と Python のほぼ完全な相乗効果が無視できないものであると確かめたことである。

Aleksey Gurtovoy は驚嘆すべき C++テンプレートメタプログラミングライブラリを書いた。これにより Boost.Python の大部分のコンパイル時のマジックが可能になった。加えて Aleksey はバグだらけの複数のコンパイラの癖に対する知識が深く、重大な局面で問題を回避できるよう気前よく時間をかけて助けてくれた。

Paul Mensonides と Vesa Karvonen は、同様に驚くべきプリプロセッサメタプログラミングライブラリを書き、それが Boost.Python で動作するよう時間と知識を使ってくれた。また Boost.Python の大部分を新しいプリプロセッサメタプログラミングツールを使うように書き直し、バグだらけで低速な C++プリプロセッサで対応が取れるよう助けてくれた。

Bruno da Silva de Oliveria は巧妙な Pyste（「Pie-Steh」と発音する）コード生成器を寄贈してくれた。

Nikolay Mladenov は `staticmethod` サポートを寄贈してくれた。

Martin Casado は、AIX の狂った動的リンクモデルに対して Boost.Python の共有ライブラリをビルド可能にするよう、厄介な問題を解決してくれた。

Achim Domma はオブジェクトラッパと、このドキュメントの HTML テンプレートを提供してくれた。Dave Hawkes は、`scope` クラスを使ってモジュール定義構文を簡単にするアイデアを提供してくれた。Pearu Pearson は現在のテストスートにあるテストケースをいくつか書いてくれた。

現バージョンの Boost.Python の開発の一部は、Lawrence Livermore National Laboratories と Lawrence Berkeley National Laboratories の Computational Crystallography Initiative の投資による。

Ullrich Koethe は類似のシステムを別に開発していた。Boost.Python v1 を見つけたとき、彼は深い見識・洞察で以って Boost.Python の強化に熱心に数え切れない時間を費やしてくれた。彼は関数の多重定義サポートの初期のバージョンについて責任を果たし、C++継承関係のリフレクションについてのサポートを書いた。彼は C++と Python の両方からのエラー報告強化を手伝ってくれ（v2 でも再度そうなればと思う）、多数の演算子のエクスポートに関数のオリジナルのサポートと多重定義による明示的な型変換を回避する方法を設計した。

Boost のメーリングリストと Python コミュニティのメンバーからは、早い段階から非常に貴重なフィードバックを得た。特に Ron Clarke、Mark Evans、Anton Gluck、Chuck Ingold、Prabhu Ramachandran、Barry Scott はまだ開発の初期段階であったにもかかわらず、勇敢にも足を踏み出して Boost.Python を使ってくれた。

Boost.Python の最初のバージョンは、開発と Boost ライブラリとしてのリリースをサポートしてくれた Dragon Systems のサポートなくしてはありえなかった。