python 2.0

Joel de Guzman

David Abrahams

Table of Contents

QuickStart

The Boost Python Library is a framework for interfacing Python and C++. It allows you to quickly and seamlessly expose C++ classes functions and objects to Python, and vice-versa, using no special tools -- just your C++ compiler. It is designed to wrap C++ interfaces non-intrusively, so that you should not have to change the C++ code at all in order to wrap it, making Boost.Python ideal for exposing 3rd-party libraries to Python. The library's use of advanced metaprogramming techniques simplifies its syntax for users, so that wrapping code takes on the look of a kind of declarative interface definition language (IDL).

Hello World

Following C/C++ tradition, let's start with the "hello, world". A C++ Function:

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

can be exposed to Python by writing a Boost.Python wrapper:

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

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

That's it. We're done. We can now build this as a shared library. The resulting DLL is now visible to Python. Here's a sample Python session:

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

Next stop... Building your Hello World module from start to finish...

Building Hello World

From Start To Finish

Now the first thing you'd want to do is to build the Hello World module and try it for yourself in Python. In this section, we will outline the steps necessary to achieve that. We will use the build tool that comes bundled with every boost distribution: **bjam**.

Note

Building without bjam

Besides bjam, there are of course other ways to get your module built. What's written here should not be taken as "the one and only way". There are of course other build tools apart from `bjam`.

Take note however that the preferred build tool for Boost.Python is bjam. There are so many ways to set up the build incorrectly. Experience shows that 90% of the "I can't build Boost.Python" problems come from people who had to use a different tool.

We will skip over the details. Our objective will be to simply create the hello world module and run it in Python. For a complete reference to building Boost.Python, check out: building.html. After this brief *bjam* tutorial, we should have built the DLLs and run a python program using the extension.

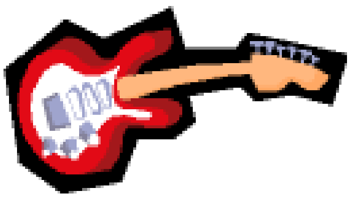
The tutorial example can be found in the directory: `libs/python/example/tutorial`. There, you can find:

- hello.cpp
- hello.py
- Jamroot

The `hello.cpp` file is our C++ hello world example. The `Jamroot` is a minimalist *bjam* script that builds the DLLs for us. Finally, `hello.py` is our Python program that uses the extension in `hello.cpp`.

Before anything else, you should have the bjam executable in your boost directory or somewhere in your path such that `bjam` can be executed in the command line. Pre-built Boost.Jam executables are available for most platforms. The complete list of Bjam executables can be found here.

Let's Jam!



Here is our minimalist Jamroot file. Simply copy the file and tweak `use-project boost` to where your boost root directory is and your OK.

The comments contained in the Jamrules file above should be sufficient to get you going.

Running bjam

bjam is run using your operating system's command line interpreter.

Start it up.

A file called user-config.jam in your home directory is used to configure your tools. In Windows, your home directory can be found by typing:

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

into a command prompt window. Your file should at least have the rules for your compiler and your python installation. A specific example of this on Windows would be:

```
#  MSVC configuration
using msvc : 8.0 ;

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

The first rule tells Bjam to use the MSVC 8.0 compiler and associated tools. The second rule provides information on Python, its version and where it is located. The above assumes that the Python installation is in `C:dev/tools\/Python`. If you have one fairly "standard" python installation for your platform, you might not need to do this.

Now we are ready... Be sure to `cd` to `libs/python/example/tutorial` where the tutorial "`hello.cpp`" and the "`Jamroot`" is situated.

Finally:

```
bjam
```

It should be building now:

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

And so on... Finally:

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

Or something similar. If all is well, you should now have built the DLLs and run the Python program.

There you go... Have fun!

Exposing Classes

Now let's expose a C++ class to Python.

Consider a C++ class/struct that we want to expose to Python:

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

We can expose this to Python by writing a corresponding Boost.Python C++ Wrapper:

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

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

Here, we wrote a C++ class wrapper that exposes the member functions `greet` and `set`. Now, after building our module as a shared library, we may use our class `World` in Python. Here's a sample Python session:

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

Constructors

Our previous example didn't have any explicit constructors. Since `World` is declared as a plain struct, it has an implicit default constructor. Boost.Python exposes the default constructor by default, which is why we were able to write

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

We may wish to wrap a class with a non-default constructor. Let us build on our previous example:

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

This time `World` has no default constructor; our previous wrapping code would fail to compile when the library tried to expose it. We have to tell `class_<World>` about the constructor we want to expose instead.

```
#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>()` exposes the constructor taking in a `std::string` (in Python, constructors are spelled "`"__init__"`").

We can expose additional constructors by passing more `init<...>`s to the `def()` member function. Say for example we have another World constructor taking in two doubles:

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

On the other hand, if we do not wish to expose any constructors at all, we may use `no_init` instead:

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

This actually adds an `__init__` method which always raises a Python RuntimeError exception.

Class Data Members

Data members may also be exposed to Python so that they can be accessed as attributes of the corresponding Python class. Each data member that we wish to be exposed may be regarded as **read-only** or **read-write**. Consider this class `Var`:

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

Our C++ `Var` class and its data members can be exposed to Python:

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

Then, in Python, assuming we have placed our Var class inside the namespace hello as we did before:

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

Note that `name` is exposed as **read-only** while `value` is exposed as **read-write**.

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

Class Properties

In C++, classes with public data members are usually frowned upon. Well designed classes that take advantage of encapsulation hide the class' data members. The only way to access the class' data is through access (getter/setter) functions. Access functions expose class properties. Here's an example:

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

However, in Python attribute access is fine; it doesn't neccessarily break encapsulation to let users handle attributes directly, because the attributes can just be a different syntax for a method call. Wrapping our `Num` class using Boost.Python:

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

And at last, in Python:

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

Take note that the class property `rovalue` is exposed as **read-only** since the `rovalue` setter member function is not passed in:

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

Inheritance

In the previous examples, we dealt with classes that are not polymorphic. This is not often the case. Much of the time, we will be wrapping polymorphic classes and class hierarchies related by inheritance. We will often have to write Boost.Python wrappers for classes that are derived from abstract base classes.

Consider this trivial inheritance structure:

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

And a set of C++ functions operating on `Base` and `Derived` object instances:

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

We've seen how we can wrap the base class `Base`:

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

Now we can inform Boost.Python of the inheritance relationship between `Derived` and its base class `Base`. Thus:

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

Doing so, we get some things for free:

1. Derived automatically inherits all of Base's Python methods (wrapped C++ member functions)
2. **If** Base is polymorphic, `Derived` objects which have been passed to Python via a pointer or reference to `Base` can be passed where a pointer or reference to `Derived` is expected.

Now, we will expose the C++ free functions `b` and `d` and `factory`:

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

Note that free function `factory` is being used to generate new instances of class `Derived`. In such cases, we use `return_value_policy<manage_new_object>` to instruct Python to adopt the pointer to `Base` and hold the instance in a new Python `Base` object until the the Python object is destroyed. We will see more of Boost.Python call policies later.

```
// Tell Python to take ownership of factory's result
def("factory", factory,
    return_value_policy<manage_new_object>());
```

Class Virtual Functions

In this section, we will learn how to make functions behave polymorphically through virtual functions. Continuing our example, let us add a virtual function to our `Base` class:

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

One of the goals of Boost.Python is to be minimally intrusive on an existing C++ design. In principle, it should be possible to expose the interface for a 3rd party library without changing it. It is not ideal to add anything to our class `Base`. Yet, when you have a virtual function that's going to be overridden in Python and called polymorphically **from C++**, we'll need to add some scaffoldings to make things work properly. What we'll do is write a class wrapper that derives from `Base` that will unintrusively hook into the virtual functions so that a Python override may be called:

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

Notice too that in addition to inheriting from `Base`, we also multiply- inherited `wrapper<Base>` (See Wrapper). The `wrapper` template makes the job of wrapping classes that are meant to overridden in Python, easier.

MSVC6/7 Workaround

If you are using Microsoft Visual C++ 6 or 7, you have to write `f` as:

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

BaseWrap's overridden virtual member function `f` in effect calls the corresponding method of the Python object through `get_override`.

Finally, exposing `Base`:

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

`pure_virtual` signals Boost.Python that the function `f` is a pure virtual function.

Note

member function and methods

Python, like many object oriented languages uses the term **methods**. Methods correspond roughly to C++'s **member functions**

Virtual Functions with Default Implementations

We've seen in the previous section how classes with pure virtual functions are wrapped using Boost.Python's class wrapper facilities. If we wish to wrap **non**-pure-virtual functions instead, the mechanism is a bit different.

Recall that in the previous section, we wrapped a class with a pure virtual function that we then implemented in C++, or Python classes derived from it. Our base class:

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

had a pure virtual function `f`. If, however, its member function `f` was not declared as pure virtual:

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

We wrap it this way:

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

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

Notice how we implemented `BaseWrap::f`. Now, we have to check if there is an override for `f`. If none, then we call `Base::f()`.

MSVC6/7 Workaround

If you are using Microsoft Visual C++ 6 or 7, you have to rewrite the line with the `*note*` as:

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

Finally, exposing:

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

Take note that we expose both `&Base::f` and `&BaseWrap::default_f`. Boost.Python needs to keep track of 1) the dispatch function `f` and 2) the forwarding function to its default implementation `default_f`. There's a special `def` function for this purpose.

In Python, the results would be as expected:

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

Calling `base.f()`:

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

Calling `derived.f()`:

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

Class Operators/Special Functions

Python Operators

C is well known for the abundance of operators. C++ extends this to the extremes by allowing operator overloading. Boost.Python takes advantage of this and makes it easy to wrap C++ operator-powered classes.

Consider a file position class `FilePos` and a set of operators that take on FilePos instances:

```
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);
```

The class and the various operators can be mapped to Python rather easily and intuitively:

```
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__
```

The code snippet above is very clear and needs almost no explanation at all. It is virtually the same as the operators' signatures. Just take note that `self` refers to FilePos object. Also, not every class `T` that you might need to interact with in an operator expression is (cheaply) default-constructible. You can use `other<T>()` in place of an actual `T` instance when writing "self expressions".

Special Methods

Python has a few more *Special Methods*. Boost.Python supports all of the standard special method names supported by real Python class instances. A similar set of intuitive interfaces can also be used to wrap C++ functions that correspond to these Python *special functions*. Example:

```
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__
    ;
```

Need we say more?

Note

What is the business of `operator<<`? Well, the method `str` requires the `operator<<` to do its work (i.e. `operator<<` is used by the method defined by `def(str(self))`.

Functions

In this chapter, we'll look at Boost.Python powered functions in closer detail. We will see some facilities to make exposing C++ functions to Python safe from potential pifalls such as dangling pointers and references. We will also see facilities that will make it even easier for us to expose C++ functions that take advantage of C++ features such as overloading and default arguments.

> *Read on...*

But before you do, you might want to fire up Python 2.2 or later and type `>>> import this`.

```
>>> import this
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!
```

Call Policies

In C++, we often deal with arguments and return types such as pointers and references. Such primitive types are rather, ummmm, low level and they really don't tell us much. At the very least, we don't know the owner of the pointer or the referenced object. No wonder languages such as Java and Python never deal with such low level entities. In C++, it's usually considered a good practice to use smart pointers which exactly describe ownership semantics. Still, even good C++ interfaces use raw references and pointers sometimes, so Boost.Python must deal with them. To do this, it may need your help. Consider the following C++ function:

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

How should the library wrap this function? A naive approach builds a Python X object around result reference. This strategy might or might not work out. Here's an example where it didn't

```
>>> x = f(y, z) # x refers to some C++ X
>>> del y
>>> x.some_method() # CRASH!
```

What's the problem?

Well, what if f() was implemented as shown below:

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

The problem is that the lifetime of result X& is tied to the lifetime of y, because the f() returns a reference to a member of the y object. This idiom is is not uncommon and perfectly acceptable in the context of C++. However, Python users should not be able to crash the system just by using our C++ interface. In this case deleting y will invalidate the reference to X. We have a dangling reference.

Here's what's happening:

1. `f` is called passing in a reference to `y` and a pointer to `z`

2. A reference to `y.x` is returned

3. `y` is deleted. `x` is a dangling reference

4. `x.some_method()` is called

5. **BOOM!**

We could copy result into a new object:

```
>>> f(y, z).set(42) # Result disappears
>>> y.x.get()       # No crash, but still bad
3.14
```

This is not really our intent of our C++ interface. We've broken our promise that the Python interface should reflect the C++ interface as closely as possible.

Our problems do not end there. Suppose Y is implemented as follows:

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

Notice that the data member `z` is held by class Y using a raw pointer. Now we have a potential dangling pointer problem inside Y:

```
>>> x = f(y, z) # y refers to z
>>> del z       # Kill the z object
>>> y.z_value() # CRASH!
```

For reference, here's the implementation of `f` again:

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

Here's what's happening:

1. `f` is called passing in a reference to `y` and a pointer to `z`

2. A pointer to `z` is held by `y`

3. A reference to `y.x` is returned
4. `z` is deleted. `y.z` is a dangling pointer
5. `y.z_value()` is called
6. `z->value()` is called
7. **BOOM!**

Call Policies

Call Policies may be used in situations such as the example detailed above. In our example, `return_internal_reference` and `with_custodian_and_ward` are our friends:

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

What are the `1` and `2` parameters, you ask?

```
return_internal_reference<1
```

Informs Boost.Python that the first argument, in our case `Y& y`, is the owner of the returned reference: `X&`. The "`1`" simply specifies the first argument. In short: "return an internal reference `X&` owned by the 1st argument `Y& y`".

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

Informs Boost.Python that the lifetime of the argument indicated by ward (i.e. the 2nd argument: `Z* z`) is dependent on the lifetime of the argument indicated by custodian (i.e. the 1st argument: `Y& y`).

It is also important to note that we have defined two policies above. Two or more policies can be composed by chaining. Here's the general syntax:

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

Here is the list of predefined call policies. A complete reference detailing these can be found here.

- **with_custodian_and_ward**: Ties lifetimes of the arguments
- **with_custodian_and_ward_postcall**: Ties lifetimes of the arguments and results
- **return_internal_reference**: Ties lifetime of one argument to that of result
- **return_value_policy<T> with T one of:**
  - **reference_existing_object**: naive (dangerous) approach
  - **copy_const_reference**: Boost.Python v1 approach
  - **copy_non_const_reference**:
  - **manage_new_object**: Adopt a pointer and hold the instance

Remember the Zen, Luke:

"Explicit is better than implicit"

"In the face of ambiguity, refuse the temptation to guess"

Overloading

The following illustrates a scheme for manually wrapping an overloaded member functions. Of course, the same technique can be applied to wrapping overloaded non-member functions.

We have here our C++ class:

```
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;
    }

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

Class X has 4 overloaded functions. We will start by introducing some member function pointer variables:

```
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;
```

With these in hand, we can proceed to define and wrap this for Python:

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

Default Arguments

Boost.Python wraps (member) function pointers. Unfortunately, C++ function pointers carry no default argument info. Take a function `f` with default arguments:

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

But the type of a pointer to the function `f` has no information about its default arguments:

```
int(*g)(int,double,char const*) = f;    // defaults lost!
```

When we pass this function pointer to the `def` function, there is no way to retrieve the default arguments:

```
def("f", f);                            // defaults lost!
```

Because of this, when wrapping C++ code, we had to resort to manual wrapping as outlined in the previous section, or writing thin wrappers:

```
// write "thin wrappers"
int f1(int x) { return f(x); }
int f2(int x, double y) { return f(x,y); }

/*...*/

    // in module init
    def("f", f);  // all arguments
    def("f", f2); // two arguments
    def("f", f1); // one argument
```

When you want to wrap functions (or member functions) that either:

- have default arguments, or
- are overloaded with a common sequence of initial arguments

BOOST_PYTHON_FUNCTION_OVERLOADS

Boost.Python now has a way to make it easier. For instance, given a function:

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

The macro invocation:

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

will automatically create the thin wrappers for us. This macro will create a class `foo_overloads` that can be passed on to `def(...)`. The third and fourth macro argument are the minimum arguments and maximum arguments, respectively. In our `foo` function the minimum number of arguments is 1 and the maximum number of arguments is 4. The `def(...)` function will automatically add all the foo variants for us:

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

BOOST_PYTHON_MEMBER_FUNCTION_OVERLOADS

Objects here, objects there, objects here there everywhere. More frequently than anything else, we need to expose member functions of our classes to Python. Then again, we have the same inconveniences as before when default arguments or overloads with a common sequence of initial arguments come into play. Another macro is provided to make this a breeze.

Like `BOOST_PYTHON_FUNCTION_OVERLOADS`, `BOOST_PYTHON_MEMBER_FUNCTION_OVERLOADS` may be used to automatically create the thin wrappers for wrapping member functions. Let's have an example:

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

The macro invocation:

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

will generate a set of thin wrappers for george's `wack_em` member function accepting a minimum of 1 and a maximum of 3 arguments (i.e. the third and fourth macro argument). The thin wrappers are all enclosed in a class named `george_overloads` that can then be used as an argument to `def(...)`:

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

See the overloads reference for details.

init and optional

A similar facility is provided for class constructors, again, with default arguments or a sequence of overloads. Remember `init<...>`? For example, given a class X with a constructor:

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

You can easily add this constructor to Boost.Python in one shot:

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

Notice the use of `init<...>` and `optional<...>` to signify the default (optional arguments).

Auto-Overloading

It was mentioned in passing in the previous section that `BOOST_PYTHON_FUNCTION_OVERLOADS` and `BOOST_PYTHON_MEMBER_FUNCTION_OVERLOADS` can also be used for overloaded functions and member functions with a common sequence of initial arguments. Here is an example:

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

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

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

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

Like in the previous section, we can generate thin wrappers for these overloaded functions in one-shot:

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

Then...

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

Notice though that we have a situation now where we have a minimum of zero (0) arguments and a maximum of 3 arguments.

Manual Wrapping

It is important to emphasize however that **the overloaded functions must have a common sequence of initial arguments**. Otherwise, our scheme above will not work. If this is not the case, we have to wrap our functions manually.

Actually, we can mix and match manual wrapping of overloaded functions and automatic wrapping through `BOOST_PYTHON_MEMBER_FUNCTION_OVERLOADS` and its sister, `BOOST_PYTHON_FUNCTION_OVERLOADS`. Following up on our example presented in the section on overloading, since the first 4 overload functins have a common sequence of initial arguments, we can use `BOOST_PYTHON_MEMBER_FUNCTION_OVERLOADS` to automatically wrap the first three of the `def`s and manually wrap just the last. Here's how we'll do this:

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

Create a member function pointers as above for both X::f overloads:

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

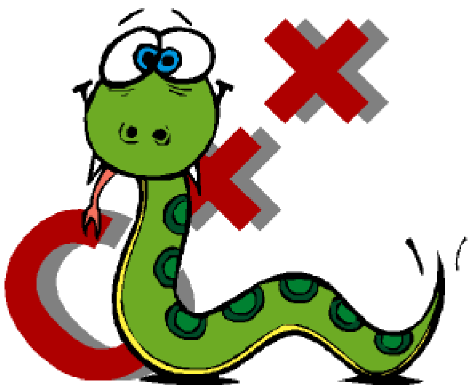
Then...

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

Object Interface

Python is dynamically typed, unlike C++ which is statically typed. Python variables may hold an integer, a float, list, dict, tuple, str, long etc., among other things. In the viewpoint of Boost.Python and C++, these Pythonic variables are just instances of class `object`. We will see in this chapter how to deal with Python objects.

As mentioned, one of the goals of Boost.Python is to provide a bidirectional mapping between C++ and Python while maintaining the Python feel. Boost.Python C++ `objects` are as close as possible to Python. This should minimize the learning curve significantly.



Basic Interface

Class `object` wraps `PyObject*`. All the intricacies of dealing with `PyObjects` such as managing reference counting are handled by the `object` class. C++ object interoperability is seamless. Boost.Python C++ `objects` can in fact be explicitly constructed from any C++ object.

To illustrate, this Python code snippet:

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

def getfunc():
   return f;
```

Can be rewritten in C++ using Boost.Python facilities this way:

```
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);
}
```

Apart from cosmetic differences due to the fact that we are writing the code in C++, the look and feel should be immediately apparent to the Python coder.

Derived Object types

Boost.Python comes with a set of derived `object` types corresponding to that of Python's:

- list
- dict
- tuple
- str
- long_
- enum

These derived `object` types act like real Python types. For instance:

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

Wherever appropriate, a particular derived `object` has corresponding Python type's methods. For instance, `dict` has a `keys()` method:

```
d.keys()
```

`make_tuple` is provided for declaring *tuple literals*. Example:

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

In C++, when Boost.Python `objects` are used as arguments to functions, subtype matching is required. For example, when a function `f`, as declared below, is wrapped, it will only accept instances of Python's `str` type and subtypes.

```
void f(str name)
{
    object n2 = name.attr("upper")();   // NAME = name.upper()
    str NAME = name.upper();            // better
    object msg = "%s is bigger than %s" % make_tuple(NAME,name);
}
```

In finer detail:

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

Illustrates that we provide versions of the str type's methods as C++ member functions.

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

Demonstrates that you can write the C++ equivalent of `"format" % x,y,z` in Python, which is useful since there's no easy way to do that in std C++.

⚠ **Beware** the common pitfall of forgetting that the constructors of most of Python's mutable types make copies, just as in Python.

Python:

```
>>> d = dict(x.__dict__)     # copies x.__dict__
>>> d['whatever'] = 3        # modifies the copy
```

C++:

```
dict d(x.attr("__dict__"));  // copies x.__dict__
d['whatever'] = 3;           // modifies the copy
```

class_<T> as objects

Due to the dynamic nature of Boost.Python objects, any `class_<T>` may also be one of these types! The following code snippet wraps the class (type) object.

We can use this to create wrapped instances. Example:

```
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);
```

Extracting C++ objects

At some point, we will need to get C++ values out of object instances. This can be achieved with the `extract<T>` function. Consider the following:

```
double x = o.attr("length"); // compile error
```

In the code above, we got a compiler error because Boost.Python `object` can't be implicitly converted to `double`s. Instead, what we wanted to do above can be achieved by writing:

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

The first line attempts to extract the "length" attribute of the Boost.Python `object`. The second line attempts to *extract* the `Vec2` object from held by the Boost.Python `object`.

Take note that we said "attempt to" above. What if the Boost.Python `object` does not really hold a `Vec2` type? This is certainly a possibility considering the dynamic nature of Python `object`s. To be on the safe side, if the C++ type can't be extracted, an appropriate exception is thrown. To avoid an exception, we need to test for extractibility:

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

The astute reader might have noticed that the `extract<T>` facility in fact solves the mutable copying problem:

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

Enums

Boost.Python has a nifty facility to capture and wrap C++ enums. While Python has no `enum` type, we'll often want to expose our C++ enums to Python as an `int`. Boost.Python's enum facility makes this easy while taking care of the proper conversions from Python's dynamic typing to C++'s strong static typing (in C++, ints cannot be implicitly converted to enums). To illustrate, given a C++ enum:

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

the construct:

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

can be used to expose to Python. The new enum type is created in the current `scope()`, which is usually the current module. The snippet above creates a Python class derived from Python's `int` type which is associated with the C++ type passed as its first parameter.

Note

what is a scope?

The scope is a class that has an associated global Python object which controls the Python namespace in which new extension classes and wrapped functions will be defined as attributes. Details can be found here.

You can access those values in Python as

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

where my_module is the module where the enum is declared. You can also create a new scope around a class:

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

// Expose X::nested as X.nested
enum_<X::nested>("nested")
    .value("red", red)
    .value("blue", blue)
    ;
```

Creating `boost::python::object` from `PyObject*`

When you want a `boost::python::object` to manage a pointer to `PyObject*` pyobj one does:

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

In this case, the `o` object, manages the `pyobj`, it won't increase the reference count on construction.

Otherwise, to use a borrowed reference:

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

In this case, `Py_INCREF` is called, so `pyobj` is not destructed when object o goes out of scope.

Embedding

By now you should know how to use Boost.Python to call your C++ code from Python. However, sometimes you may need to do the reverse: call Python code from the C++-side. This requires you to *embed* the Python interpreter into your C++ program.

Currently, Boost.Python does not directly support everything you'll need when embedding. Therefore you'll need to use the Python/C API to fill in the gaps. However, Boost.Python already makes embedding a lot easier and, in a future version, it may become unnecessary to touch the Python/C API at all. So stay tuned... 🙂

Building embedded programs

To be able to embed python into your programs, you have to link to both Boost.Python's as well as Python's own runtime library.

Boost.Python's library comes in two variants. Both are located in Boost's `/libs/python/build/bin-stage` subdirectory. On Windows, the variants are called `boost_python.lib` (for release builds) and `boost_python_debug.lib` (for debugging). If you can't find the libraries, you probably haven't built Boost.Python yet. See Building and Testing on how to do this.

Python's library can be found in the `/libs` subdirectory of your Python directory. On Windows it is called pythonXY.lib where X.Y is your major Python version number.

Additionally, Python's `/include` subdirectory has to be added to your include path.

In a Jamfile, all the above boils down to:

```
projectroot c:\projects\embedded_program ; # location of the program

# bring in the rules for python
SEARCH on python.jam = $(BOOST_BUILD_PATH) ;
include python.jam ;

exe embedded_program # name of the executable
  : #sources
     embedded_program.cpp
  : # requirements
     <find-library>boost_python <library-path>c:\boost\libs\python
  $(PYTHON_PROPERTIES)
    <library-path>$(PYTHON_LIB_PATH)
    <find-library>$(PYTHON_EMBEDDED_LIBRARY) ;
```

Getting started

Being able to build is nice, but there is nothing to build yet. Embedding the Python interpreter into one of your C++ programs requires these 4 steps:

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

2. Call Py_Initialize() to start the interpreter and create the `__main__` module.

3. Call other Python C API routines to use the interpreter.

Note

Note that at this time you must not call Py_Finalize() to stop the interpreter. This may be fixed in a future version of boost.python.

(Of course, there can be other C++ code between all of these steps.)

Now that we can embed the interpreter in our programs, lets see how to put it to use...

Using the interpreter

As you probably already know, objects in Python are reference-counted. Naturally, the `PyObjects` of the Python C API are also reference-counted. There is a difference however. While the reference-counting is fully automatic in Python, the Python C API requires you to do it by hand. This is messy and especially hard to get right in the presence of C++ exceptions. Fortunately Boost.Python provides the handle and object class templates to automate the process.

Running Python code

Boost.python provides three related functions to run Python code from C++.

```
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 evaluates the given expression and returns the resulting value. exec executes the given code (typically a set of statements) returning the result, and exec_file executes the code contained in the given file.

The `globals` and `locals` parameters are Python dictionaries containing the globals and locals of the context in which to run the code. For most intents and purposes you can use the namespace dictionary of the `__main__` module for both parameters.

Boost.python provides a function to import a module:

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

import imports a python module (potentially loading it into the running process first), and returns it.

Let's import the `__main__` module and run some Python code in its namespace:

```
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);
```

This should create a file called 'hello.txt' in the current directory containing a phrase that is well-known in programming circles.

Manipulating Python objects

Often we'd like to have a class to manipulate Python objects. But we have already seen such a class above, and in the previous section: the aptly named `object` class and its derivatives. We've already seen that they can be constructed from a `handle`. The following examples should further illustrate this fact:

```
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"]);
```

Here we create a dictionary object for the `__main__` module's namespace. Then we assign 5 squared to the result variable and read this variable from the dictionary. Another way to achieve the same result is to use eval instead, which returns the result directly:

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

Exception handling

If an exception occurs in the evaluation of the python expression, error_already_set is thrown:

```
try
{
    object result = eval("5/0");
    // execution will never get here:
    int five_divided_by_zero = extract<int>(result);
}
catch(error_already_set const &)
{
    // handle the exception in some way
}
```

The `error_already_set` exception class doesn't carry any information in itself. To find out more about the Python exception that occurred, you need to use the exception handling functions of the Python C API in your catch-statement. This can be as simple as calling PyErr_Print() to print the exception's traceback to the console, or comparing the type of the exception with those of the standard exceptions:

```
catch(error_already_set const &)
{
    if (PyErr_ExceptionMatches(PyExc_ZeroDivisionError))
    {
        // handle ZeroDivisionError specially
    }
    else
    {
        // print all other errors to stderr
        PyErr_Print();
    }
}
```

(To retrieve even more information from the exception you can use some of the other exception handling functions listed here.)

Iterators

In C++, and STL in particular, we see iterators everywhere. Python also has iterators, but these are two very different beasts.

C++ iterators:

- C++ has 5 type categories (random-access, bidirectional, forward, input, output)
- There are 2 Operation categories: reposition, access
- A pair of iterators is needed to represent a (first/last) range.

Python Iterators:

- 1 category (forward)
- 1 operation category (next())
- Raises StopIteration exception at end

The typical Python iteration protocol: **`for y in x...`** is as follows:

```
iter = x.__iter__()         # get iterator
try:
    while 1:
    y = iter.next()         # get each item
    ...                     # process y
except StopIteration: pass  # iterator exhausted
```

Boost.Python provides some mechanisms to make C++ iterators play along nicely as Python iterators. What we need to do is to produce appropriate `__iter__` function from C++ iterators that is compatible with the Python iteration protocol. For example:

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

Or for use in class_<>:

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

range

We can create a Python savvy iterator using the range function:

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

Here, start/finish may be one of:

- member data pointers
- member function pointers
- adaptable function object (use Target parameter)

iterator

- iterator<T, Policies>()

Given a container T, iterator is a shortcut that simply calls range with &T::begin, &T::end.

Let's put this into action... Here's an example from some hypothetical bogon Particle accelerator code:

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

Now, our C++ Wrapper:

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

stl_input_iterator

So far, we have seen how to expose C++ iterators and ranges to Python. Sometimes we wish to go the other way, though: we'd like to pass a Python sequence to an STL algorithm or use it to initialize an STL container. We need to make a Python iterator look like an STL iterator. For that, we use `stl_input_iterator<>`. Consider how we might implement a function that exposes `std::list<int>::assign()` to Python:

```
template<typename T>
void list_assign(std::list<T>& l, object o) {
    // Turn a Python sequence into an STL input range
    stl_input_iterator<T> begin(o), end;
    l.assign(begin, end);
}

// Part of the wrapper for list<int>
class_<std::list<int> >("list_int")
    .def("assign", &list_assign<int>)
    // ...
    ;
```

Now in Python, we can assign any integer sequence to `list_int` objects:

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

Exception Translation

All C++ exceptions must be caught at the boundary with Python code. This boundary is the point where C++ meets Python. Boost.Python provides a default exception handler that translates selected standard exceptions, then gives up:

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

Users may provide custom translation. Here's an example:

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

General Techniques

Here are presented some useful techniques that you can use while wrapping code with Boost.Python.

Creating Packages

A Python package is a collection of modules that provide to the user a certain functionality. If you're not familiar on how to create packages, a good introduction to them is provided in the Python Tutorial.

But we are wrapping C++ code, using Boost.Python. How can we provide a nice package interface to our users? To better explain some concepts, let's work with an example.

We have a C++ library that works with sounds: reading and writing various formats, applying filters to the sound data, etc. It is named (conveniently) `sounds`. Our library already has a neat C++ namespace hierarchy, like so:

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

We would like to present this same hierarchy to the Python user, allowing him to write code like this:

```
import sounds.filters
sounds.filters.echo(...) # echo is a C++ function
```

The first step is to write the wrapping code. We have to export each module separately with Boost.Python, like this:

```
/* file core.cpp */
BOOST_PYTHON_MODULE(core)
{
    /* export everything in the sounds::core namespace */
    ...
}

/* file io.cpp */
BOOST_PYTHON_MODULE(io)
{
    /* export everything in the sounds::io namespace */
    ...
}

/* file filters.cpp */
BOOST_PYTHON_MODULE(filters)
{
    /* export everything in the sounds::filters namespace */
    ...
}
```

Compiling these files will generate the following Python extensions: `core.pyd`, `io.pyd` and `filters.pyd`.

Note

The extension `.pyd` is used for python extension modules, which are just shared libraries. Using the default for your system, like `.so` for Unix and `.dll` for Windows, works just as well.

Now, we create this directory structure for our Python package:

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

The file `__init__.py` is what tells Python that the directory `sounds/` is actually a Python package. It can be a empty file, but can also perform some magic, that will be shown later.

Now our package is ready. All the user has to do is put `sounds` into his PYTHONPATH and fire up the interpreter:

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

Nice heh?

This is the simplest way to create hierarchies of packages, but it is not very flexible. What if we want to add a *pure* Python function to the filters package, for instance, one that applies 3 filters in a sound object at once? Sure, you can do this in C++ and export it, but why not do so in Python? You don't have to recompile the extension modules, plus it will be easier to write it.

If we want this flexibility, we will have to complicate our package hierarchy a little. First, we will have to change the name of the extension modules:

```
/* file core.cpp */
BOOST_PYTHON_MODULE(_core)
{
    ...
    /* export everything in the sounds::core namespace */
}
```

Note that we added an underscore to the module name. The filename will have to be changed to `_core.pyd` as well, and we do the same to the other extension modules. Now, we change our package hierarchy like so:

```
sounds/
    __init__.py
    core/
        __init__.py
        core.pyd
____filters/
________\_init__.py
        filters.pyd
____io/
________\_init__.py
        _io.pyd
```

Note that we created a directory for each extension module, and added a __init__.py to each one. But if we leave it that way, the user will have to access the functions in the core module with this syntax:

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

which is not what we want. But here enters the `__init__.py` magic: everything that is brought to the `__init__.py` namespace can be accessed directly by the user. So, all we have to do is bring the entire namespace from `_core.pyd` to `core/__init__.py`. So add this line of code to `sounds/core/__init__.py`:

```
from _core import *
```

We do the same for the other packages. Now the user accesses the functions and classes in the extension modules like before:

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

with the additional benefit that we can easily add pure Python functions to any module, in a way that the user can't tell the difference between a C++ function and a Python function. Let's add a *pure* Python function, `echo_noise`, to the `filters` package. This function applies both the `echo` and `noise` filters in sequence in the given `sound` object. We create a file named `sounds/filters/echo_noise.py` and code our function:

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

Next, we add this line to `sounds/filters/__init__.py`:

```
from echo_noise import echo_noise
```

And that's it. The user now accesses this function like any other function from the `filters` package:

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

Extending Wrapped Objects in Python

Thanks to Python's flexibility, you can easily add new methods to a class, even after it was already created:

```
>>> class C(object): pass
>>>
>>> # a regular function
>>> def C_str(self): return 'A C instance!'
>>>
>>> # now we turn it in a member function
>>> C.__str__ = C_str
>>>
>>> c = C()
>>> print c
A C instance!
>>> C_str(c)
A C instance!
```

Yes, Python rox.

We can do the same with classes that were wrapped with Boost.Python. Suppose we have a class `point` in C++:

```
class point {...};

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

If we are using the technique from the previous session, Creating Packages, we can code directly into `geom/__init__.py`:

```
from _geom import *

# a regular function
def point_str(self):
    return str((self.x, self.y))

# now we turn it into a member function
point.__str__ = point_str
```

All point instances created from C++ will also have this member function! This technique has several advantages:

- Cut down compile times to zero for these additional functions
- Reduce the memory footprint to virtually zero
- Minimize the need to recompile
- Rapid prototyping (you can move the code to C++ if required without changing the interface)

You can even add a little syntactic sugar with the use of metaclasses. Let's create a special metaclass that "injects" methods in other classes.

```
# The one Boost.Python uses for all wrapped classes.
# You can use here any class exported by Boost instead of "point"
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)

# inject some methods in the point foo
class more_point(injector, point):
    def __repr__(self):
        return 'Point(x=%s, y=%s)' % (self.x, self.y)
    def foo(self):
        print 'foo!'
```

Now let's see how it got:

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

Another useful idea is to replace constructors with factory functions:

```
_point = point

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

In this simple case there is not much gained, but for constructurs with many overloads and/or arguments this is often a great simplification, again with virtually zero memory footprint and zero compile-time overhead for the keyword support.

Reducing Compiling Time

If you have ever exported a lot of classes, you know that it takes quite a good time to compile the Boost.Python wrappers. Plus the memory consumption can easily become too high. If this is causing you problems, you can split the class_ definitions in multiple files:

```
/* file point.cpp */
#include <point.h>
#include <boost/python.hpp>

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

/* file triangle.cpp */
#include <triangle.h>
#include <boost/python.hpp>

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

Now you create a file `main.cpp`, which contains the `BOOST_PYTHON_MODULE` macro, and call the various export functions inside it.

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

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

Compiling and linking together all this files produces the same result as the usual approach:

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

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

but the memory is kept under control.

This method is recommended too if you are developing the C++ library and exporting it to Python at the same time: changes in a class will only demand the compilation of a single cpp, instead of the entire wrapper code.

Note

If you're exporting your classes with Pyste, take a look at the `--multiple` option, that generates the wrappers in various files as demonstrated here.

Note

This method is useful too if you are getting the error message *"fatal error C1204:Compiler limit:internal structure overflow"* when compiling a large source file, as explained in the FAQ.