BOOST
CONSULTING

Hybrid Development with Boost.Python (and More!)

Dynamic Language Interoperability

Dave Abrahams, Boost Consulting

01/17/11

Agenda

Agenda

- Talk about Boost.Python
- Promote Hybrid Development

Agenda

- Talk about Boost.Python
- Promote Hybrid Development

- Hidden Agenda

Agenda

- Talk about Boost.Python
- Promote Hybrid Development

- Hidden Agenda
  - Domain Specific Embedded Languages

Agenda

- Talk about Boost.Python
- Promote Hybrid Development

- Hidden Agenda
  - Domain Specific Embedded Languages
  - Promote Boost.Langbinding Project

Python vs. C++

Different Approaches…

- Interpreted
- Dynamic Typing
- Easily Ported
- Everything at runtime
- Flexible
- Broad library selection

- Compiled
- Statically Typed
- Powerful Compiler
- Much at compile-time
- Efficient
- Libs focused on core functionality

…But Similar Idioms

- 'C' family control structures and syntax.
- Classes, functions, modular design
- Operator Overloading:
  - Expressiveness
  - Science/Math
- Emphasis on Libraries
- High-level:
  - Containers/Iterators
  - Exception handling
- Multiparadigm:
  - OOP
  - Functional Programming
  - Generic Programming

…But Similar Idioms

- 'C' family control structures and syntax.
- Classes, functions, modular design
- Operator Overloading:
  - Expressiveness
  - Science/Math
- Emphasis on Libraries

- High-level:
  - Containers/Iterators
  - Exception handling
- Multiparadigm:
  - OOP
  - Functional Programming
  - Generic Programming

Two great tastes that taste great together

Python and C++ together

- **Extending** Python with C++ "extension modules"
  - Shared library plug-ins
  - Experimentation, prototyping, systems integration
- **Embedding** Python in a C++ program
  - Link Python into executable
  - Scriptability, access to broad Python library
  - Usually combined with extending

Plugins vs. Large-Scale Development

- Plugins
  - Loaded with dlopen/LoadLibrary
  - Single Entry Point
  - Total Isolation
- Large-Scale Development
  - Multiple dynamic library developers
  - Objects passed between modules
  - Shared symbol space

Low-level Python/C++

- Traditional approach: use Python 'C' API
  - Everything is a PyObject*
  - Manual reference counting
  - Type system underdocumented

- Typical result
  - Minimalist extension module with Python front-end → crippled extension classes
  - Boilerplate code repetition (args/returns)
  - Insecure (EH, pointers, and overflow)

Hello, Python 'C' API World

```
extern "C" // all Python interactions use 'C' linkage and calling convention
{
    // wrapper to handle argument/result conversion and checking
    PyObject* greet_wrap(PyObject* args, PyObject* keywords)
    {
        int x;
        if (PyArg_ParseTuple(args, "i", &x))    // extract/check arguments
        {
            char const* result = greet(x);      // invoke wrapped function
            return PyString_FromString(result); // convert to Python
        }
        return 0;                               // error occurred
    }


    // table of wrapped functions to be exposed by the module
    static PyMethodDef methods[] = {
       { "greet", greet_wrap, METH_VARARGS,
              "return one of 3 parts of a greeting" }
      , { NULL, NULL, 0, NULL } // sentinel
    };


    // module initialization function
    DL_EXPORT init_hello()
```

High-level Python/C++

- Interface-specification file (IDL)➔ C code (ILU)
- C++-like IDL➔ Python 'C' API code (SIP)
- Full C++ ➔ Python 'C' API code (SWIG, GRAD)
- C++ wrappers for Python API (CXX)
- Introspective C++ IDL library framework (Boost)

Hello, Boost.Python World!

- C++ Interface:

```
char const* greet(int x);
```

- C++ Wrapper:

```
BOOST_PYTHON_MODULE_INIT(hello)
{
   def("greet", greet);
}
```

- Python:

```
>>> for x in range(3):
...       print hello.greet(x),
...
hello, Boost.Python world!
```

Hello, Boost.Python World!

- C++ Interface:

```
char const* greet(int x)
{
   static char const* const msgs[] = {
         "hello,", "Boost.Python", "world!"
         };
   return msgs[x];
}
```

- C++ Wrapper:

```
BOOST_PYTHON_MODULE_INIT(hello)
{
   def("greet", greet);
}
```

- Python:

```
>>> for x in range(3):
...       print hello.greet(x),
...
hello, Boost.Python world!
```

Hello, Boost.Python World!

- C++ Interface:

```
char const* greet(int x);
```

- C++ Wrapper:

```
BOOST_PYTHON_MODULE_INIT(hello)
{
   def("greet", greet);
}
```

- Python:

```
>>> for x in range(3):
...       print hello.greet(x),
...
hello, Boost.Python world!
```

Boost.Python – Design Goals

- Reflect C++ interfaces into Python
- Non-intrusive
- Do it all in C++ with minimal intervention
- Insulate C++ users from Python 'C' API
- Insulate Python users from C++ (crashes)
- Respect Both C++ and Python idioms
- Support component-based development

Boost.Python Wrapper DSEL

image.hpp

```
namespace image
{
    class Canvas
    {
       void erase();
       ...
    };

    std::auto_ptr<canvas>
    create(int h, int v);
```

wrap_image.cpp

```
using namespace image;


BOOST_PYTHON_MODULE("image")
{
      class_<Canvas>("Canvas")
         .def("erase"
           ,&Canvas::erase)
        ;


    def("create", &create);
}
```

Boost.Python Wrapper DSEL

image.hpp

```
namespace image
{
    class Canvas
    {
       void erase();
       ...
    };

    std::auto_ptr<canvas>
    create(int h, int v);
```

wrap_image.cpp

```
using namespace image;


BOOST_PYTHON_MODULE("image")
{
      class_<Canvas>("Canvas")
        .def("erase"
          ,&Canvas::erase)
        ;


    def("create", &create);
}
```

Boost.Python Wrapper DSEL

image.hpp

```
namespace image
{
    class Canvas
    {
       void erase();
       ...
    };

    std::auto_ptr<canvas>
    create(int h, int v);
```

wrap_image.cpp

```
using namespace image;


BOOST_PYTHON_MODULE("image")
{
      class_<Canvas>("Canvas")
         .def("erase"
           ,&Canvas::erase)
        ;


    def("create", &create);
}
```

Boost.Python Wrapper DSEL

image.hpp

```
namespace image
{
    class Canvas
    {
       void erase();
       ...
    };

    std::auto_ptr<canvas>
    create(int h, int v);
```

wrap_image.cpp

```
using namespace image;


BOOST_PYTHON_MODULE("image")
{
      class_<Canvas>("Canvas")
         .def("erase"
            ,&Canvas::erase)
          ;


    def("create", &create);
}
```

Boost.Python Wrapper DSEL

image.hpp

```
namespace image
{
    class Canvas
    {
       void erase();
       ...
    };

    std::auto_ptr<canvas>
    create(int h, int v);
```

wrap_image.cpp

```
using namespace image;


BOOST_PYTHON_MODULE("image")
{
      class_<Canvas>("Canvas")
        .def("erase"
          ,&Canvas::erase)
        ;

    def("create", &create);
}
```

Language-Neutral Wrapper DSEL

image.hpp

```
namespace image
{
    class Canvas
    {
       void erase();
       ...
    };

    std::auto_ptr<canvas>
    create(int h, int v);
```

wrap_image.cpp

```
using namespace image;
void init()
{
  module("image")
  [
      class_<Canvas>("Canvas")

         .def("erase"
           ,&Canvas::erase),


        def("create", &create)

    ];
}
```

Language-Neutral Wrapper DSEL

image.hpp

```
namespace image
{
    class Canvas
    {
       void erase();
       ...
    };

    std::auto_ptr<canvas>
    create(int h, int v);
```

wrap_image.cpp

```
using namespace image;
void init()
{
  module("image")
  [
      class_<Canvas>("Canvas")

         .def("erase"
           ,&Canvas::erase),


        def("create", &create)

    ];
}
```

Language-Neutral Wrapper DSEL

image.hpp

```
namespace image
{
    class Canvas
    {
       void erase();
       ...
    };

    std::auto_ptr<canvas>
    create(int h, int v);
```

wrap_image.cpp

```
using namespace image;
void init()
{
  module("image")
  [
      class_<Canvas>("Canvas")

        .def("erase"
          ,&Canvas::erase),


      def("create", &create)

    ];
}
```

Language-Neutral Wrapper DSEL

image.hpp

```
namespace image
{
    class Canvas
    {
       void erase();
       ...
    };

    std::auto_ptr<canvas>
    create(int h, int v);
```

wrap_image.cpp

```
using namespace image;
void init()
{
  module("image")
  [
      class_<Canvas>("Canvas")

          def("erase"
           ,&Canvas::erase),


        def("create", &create)

    ];
}
```

Language-Neutral Wrapper DSEL

image.hpp

```
namespace image
{
    class Canvas
    {
        void erase();
        ...
    };

    std::auto_ptr<canvas>
    create(int h, int v);
```

wrap_image.cpp

```
using namespace image;
void init()
{
  module("image")
  [
      class_<Canvas>("Canvas")

          .def("erase"
            ,&Canvas::erase),


          def("create", &create)

      ];
}
```

Language-Neutral Wrapper DSEL

image.hpp

```
namespace image
{
    class Canvas
    {
       void erase();
       ...
    };

    std::auto_ptr<canvas>
    create(int h, int v);
```

wrap_image.cpp

```
using namespace image;
void init()
{
    module("image")
        class_<Canvas>("Canvas")

            .def("erase"
              ,&Canvas::erase),


            def("create", &create)

        ];
}
```

Boost.Langbinding Only

Basic Function Wrapping Interface

def(*name***,** [*member-*]*function-pointer***)**

- Member/free function duality

  R (X::*)(A1) ≡ R (*)(X&, A1)

- Overloading
  - Use multiple **def**s with same name
  - Best match algorithm from Luabind

Basic Function Wrapping Interface

def(*name***, [***member-***]*function-pointer***)**

- Member/free function duality

  R (X::*)(A1) ≡ R (*)(X&, A1)

- Overloading
  - Use multiple **def**s with same name
  - Best match algorithm à la Luabind

Boost.Langbinding Only

Exposing Classes

- C++ Interface:

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

- C++ Wrapper:

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

- Python:

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

Exposing Classes

- C++ Interface:

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

- C++ Wrapper:

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

- Python:

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

Wrapping Classes

Wrapping Classes

- Wrap *class* (default constructible):

  class_<*class***>(***name***)**

Wrapping Classes

- Wrap *class* (default constructible):

  class_<*class*>(*name*)

- Specify constructor arguments a_1, a_2...:

  class_<*class*>(*name* **, init**<a_1, a_2...>**()**)

Wrapping Classes

- Wrap *class* (default constructible):

  class_<*class*>(*name*)

- Specify constructor arguments a_1, a_2...:

  class_<*class*>(*name* **, init**<a_1, a_2...>**()**)

- Specify non-constructibility (abstract):

  class_<*class*>(*name* **, no_init()**)

Wrapping Classes

- Wrap *class* (default constructible):

  class_<*class*>(*name*)

- Specify constructor arguments a_1, a_2...:

  class_<*class*>(*name* , **init**<a_1, a_2...>())

- Specify non-constructibility (abstract):

  class_<*class*>(*name* , **no_init**())

- Specify Bases b_1, b_2...:

Wrapping Classes

- Wrap *class* (default constructible):

  class_<*class*>(*name*)

- Specify constructor arguments a_1, a_2...:

  class_<*class*>(*name* , **init**<a_1, a_2...>())

- Specify non-constructibility (abstract):

  class_<*class*>(*name* , **no_init**())

- Specify Bases b_1, b_2... (langbinding):

Boost.Python Object Model

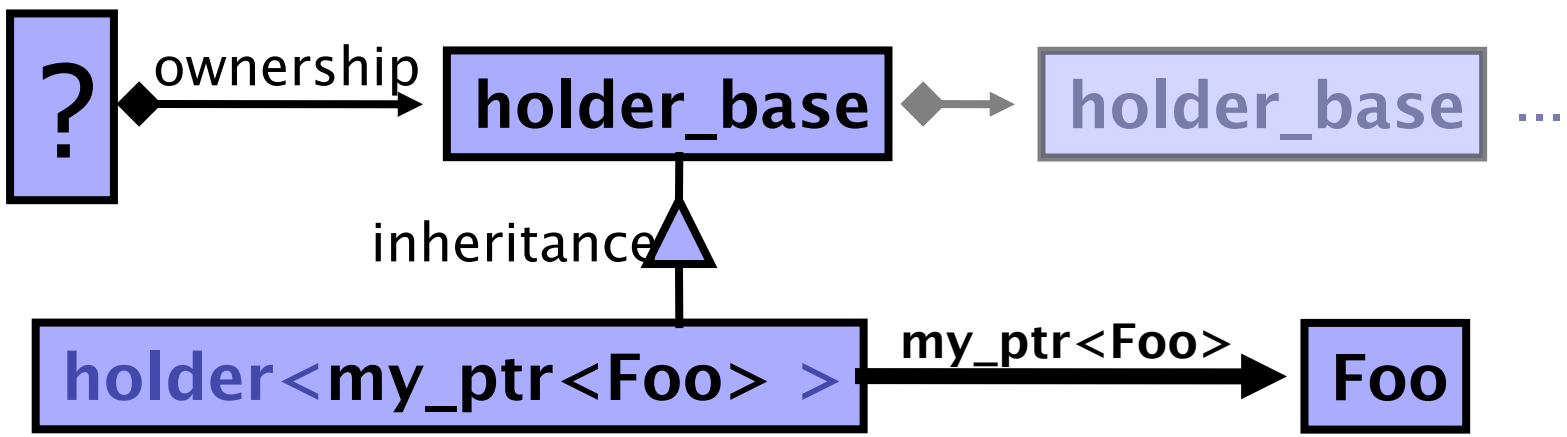
Python Instance data area contains one or more **Holders** that

- maintain C++ instance data
- Can hold by value or (smart) pointer
- Must answer the question "do you hold an instance of this type?"

Specifying a holder pointer

- Wrapped as:

```
class_<Foo, my_ptr<Foo> >("Foo")
```



- Upshot: Python **Foo** instances are convertible to **my_ptr<Foo>**

Class Members

- Constructors, functions, and data

```
class_<X>("X")      // implied default ctor
   .def(init<int>()) // additional ctor

   .def("foo", &X::foo1)// overloaded foo
   .def("foo", &X::foo2)

   .def("val", &X::val) // data member
   ;
```

Overridable Virtual Functions

```
class Base
{
 protected:
   virtual int f(int);
};
```

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

```
struct BaseWrap
  : Base, polymorphic<Base>
{
    virtual int f(int x)
    {
      if (
            override f =
        find_override("f")
       )
          return f(x);
      else
          return Base::f(x);
    }

    int default_f(int x)
    {
        return Base::f(x);
    }
};
```

Exposing Operators

```
class fpos { … };

fpos   operator+(fpos, int);
fpos   operator+(int, fpos);

int operator-(fpos, fpos);
fpos   operator-(fpos, int);

fpos& operator+=(fpos&, int);
fpos& operator-=(fpos&, int);

bool   operator<(fpos, fpos);
```

```
class_<fpos>()

    .def(self + int())
    .def(int() + self)

    .def(self - self)
    .def(self - int())

    .def(self += int())
    .def(self -= other<int>())

    .def(self < self)
```

Other "Special Functions"

- C++ Interface:

```
class Num
{
   operator double() const;
};

Rational pow(Num, Num);
Rational abs(Num);

ostream& operator<<(
          ostream&, Num);
```

- C++ Wrapper:

```
class_<Num>()
  .def(float_(self)) // __float__




  .def(pow(self,self)) // __pow__
  .def(abs(self))      // __abs__

  .def(str(self))      // __str__
  ;
```

Properties

- C++ Interface:

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

- C++ Wrapper:

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

- Python:

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

Call Policies

- Problem: raw pointers and references don't tell us much:

```
X& f1(Y& y);
```

- Naïve approach builds a Python X object around result reference:

```
>>> x = f1(y)      # x refers to some C++ X
>>> del y
>>> x.some_method() # CRASH!
```

- What's the problem?

Call Policies

- Semantics of f() tie lifetime of result to y

```
X& f1(Y& y)
{
    return y.x;
}
```

- Could copy result into new object (c.f. v1)

```
>>> f1(y).set(42) # Result disappears
>>> y.x.get()              # No crash; still bad
3.14
```

- Doesn't reflect C++ interface

Call Policies

- Stored pointers

```
struct Y
{
    Y(Z* z) : z(z) {}

    int z_value() { return z->value(); }

    Z* z;
};
```

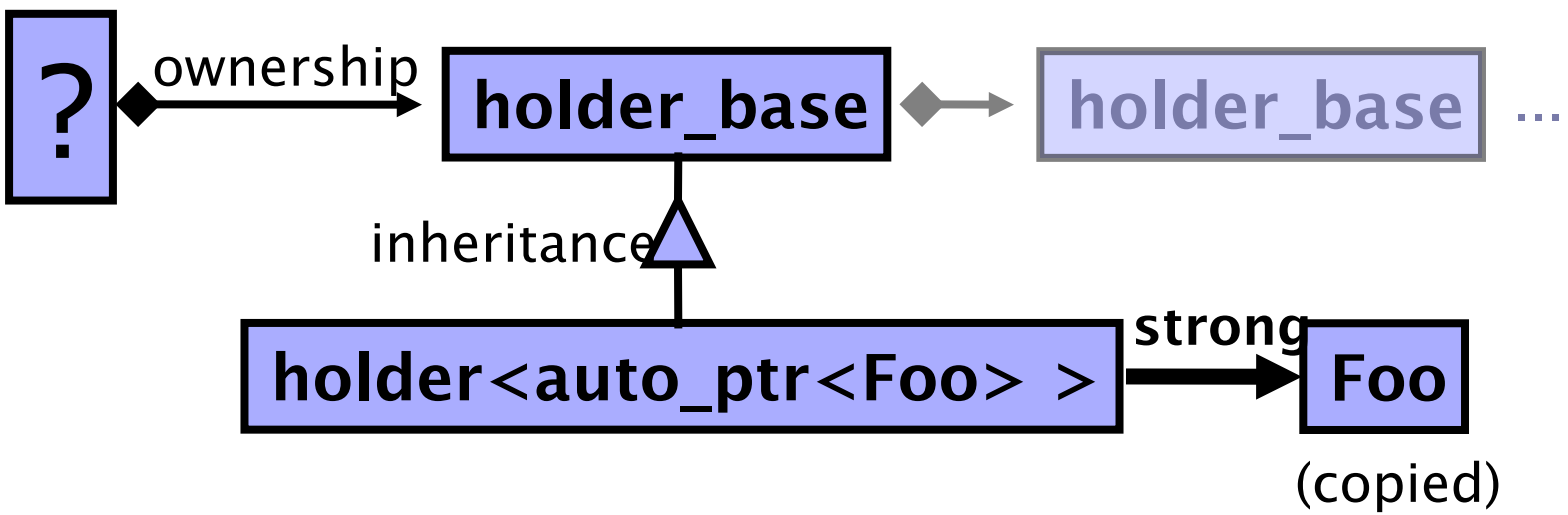
- More problems:

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

Wrapped Class Object Model

- Safe reference return behavior:

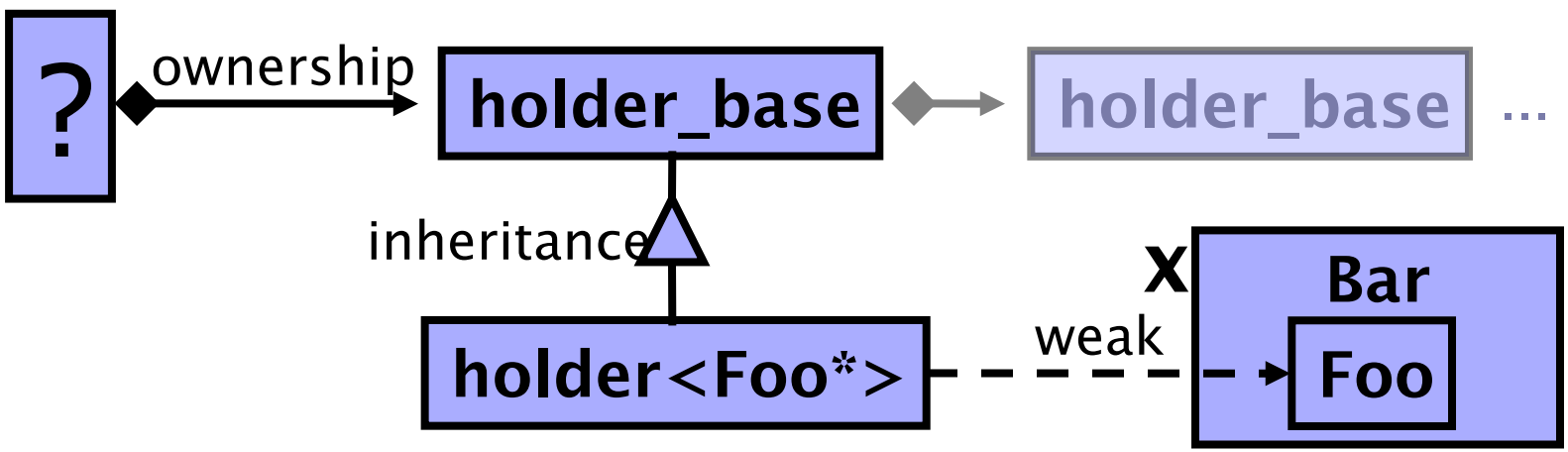
  Foo const& get(Bar& x);



Wrapped Class Object Model

- Sometimes you want this:

  Foo const& get(Bar& x);



Call Policies – Application

- C++ Implementation

```
X& get_x(Y& y)
{
   return y.x;
}
```

- C++ Wrapping code:

```
def(
   “get_x”, get_x
  , return_internal_reference<1>() );
```

Call Policies – Application

- C++ Implementation

```
X& Y::get_x()
{
   return this->x;
}
```

- C++ Wrapping code:

```
class_<Y>("Y")
    .def(
        "get_x", &Y::get_x
      , return_internal_reference<1>()
    )
;
```

Call Policies – Application

- C++ Implementation

```
X& Y::get_x()
{
   return this->x;
}
```

- C++ Wrapping code:

```
class_<Y>("Y")
    .def(
        "get_x", &Y::get_x
      , return_internal_reference<>()
    )
;
```

Call Policies – Application

- C++ Implementation

```
void Y::set_z(Z* z)
{
    this->z = z;
}
```

- C++ Wrapping code:

```
def(
    "set_z", &Y::set_z,
     with_custodian_and_ward<1,2> >() );
```

Chaining Call Policies

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

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

Chaining Call Policies

- C++ Implementation

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



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

Chaining Call Policies

- C++ Implementation

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



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

Chaining Call Policies

- C++ Implementation

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



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

Chaining Call Policies

- C++ Implementation

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

- C++ Wrapping code:

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

Chaining Call Policies

- C++ Implementation

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

- C++ Wrapping code:

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

Chaining Call Policies

- C++ Implementation

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

- C++ Wrapping code:

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

Chaining Call Policies

- C++ Implementation

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

- C++ Wrapping code:

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

- Ideally, relationships are documented!

Call Policies – Models

- **with_custodian_and_ward** – ties lifetimes of args
- **with_custodian_and_ward_postcall** – ties lifetimes of args (and result)
- **return_internal_reference** - ties lifetime of one argument to that of result
- **return_value_policy**<T> with T one of:
  - **reference_existing_object** – naïve (dangerous) approach
  - **copy_const_reference** – naïve (safe) approach
  - **copy_non_const_reference**
  - **manage_new_object** – adopt-a-pointer

Langbinding Policies - Sweet!

Langbinding Policies – Sweet!

Foo const& get(**Bar&**);

def(“get”, &get, **return_reference_to(_1)**);

Langbinding Policies – Sweet!

```
Foo const& get(Bar&);
def( “get”, &get, return_reference_to(_1) );


void acquire(Foo*);
def( “acquire”, &acquire, adopt(_1) );

Bar* Bar::instance()
{
    static Bar x;
```

Langbinding Policies – Sweet!

```
Foo const& get(Bar&);
def( “get”, &get, return_reference_to(_1) );


void acquire(Foo*);
def( “acquire”, &acquire, adopt(_1) );

Bar* Bar::instance()
{
    static Bar x;
    return &x;
```

Langbinding Policies – Sweet!

```
Foo const& get(Bar&);
def( "get", &get, return_reference_to(_1) );


void acquire(Foo*);
def( "acquire", &acquire, adopt(_1) );

Bar* Bar::instance()
{
    static Bar x;
    return &x;
}
```

Langbinding Policies – Sweet!

```
Foo const& get(Bar&);
def( "get", &get, return_reference_to(_1) );


void acquire(Foo*);
def( "acquire", &acquire, adopt(_1) );

Bar* Bar::instance()
{
    static Bar x;
    return &x;
}
def( "instance", &Bar::instance
```

Langbinding Policies – Sweet!

```
Foo const& get(Bar&);
def( “get”, &get, return_reference_to(_1) );


void acquire(Foo*);
def( “acquire”, &acquire, adopt(_1) );

Bar* Bar::instance()
{
    static Bar x;
    return &x;
}
def( “instance”, &Bar::instance
    , return_raw_reference() );
```

Langbinding Policies – Sweet!

```
Foo const& get(Bar&);
def( “get”, &get, return_reference_to(_1) );


void acquire(Foo*);
def( “acquire”, &acquire, adopt(_1) );

Bar* Bar::instance()
{
    static Bar x;
    return &x;
}
def( “instance”, &Bar::instance
    , return_raw_reference() );
```

Default Arguments

```
int f(int, double = 3.14, char const* = “hello”);
int (*g)(int,double,char const*) = f; // defaults lost
int x = g(3);                                          // error!
```

```
// write “thin wrappers”
int f1(int x) { f(x); }

int f2(int x, double y) { f(x,y); }
…
   def(“f”,f); def(“f”, f2); def(“f”, f1); // in module init
```

Default Arguments

- Boost.Python wraps (member) function pointers

- C++ function pointers carry no default arg info:

```
int f(int, double = 3.14, char const* = "hello");
int (*g)(int,double,char const*) = f; // defaults lost
int x = g(3);                                    // error!
```

- C++ Wrapping code (old way):

```
// write "thin wrappers"
int f1(int x) { f(x); }

int f2(int x, double y) { f(x,y); }
…
   def("f",f); def("f", f2); def("f", f1); // in module init
```

Default Arguments

- C++ Wrapping code (new way):

```
// Macro declares f_defaults
BOOST_PYTHON_FUNCTION_GENERATOR(f_defaults, f, 1, 3)
    …
    def("f", f, f_defaults()); // In module init
```

- Similarly for classes

```
BOOST_PYTHON_MEM_FUN_GENERATOR(m_defaults, m, 0, 7)
  …
  class_<X>("X", init<std::string, optional<int, int> >)
      .def("m", &X::m, m_defaults())
      ;
```

Python Object Interface

- Class **object** wraps PyObject*
- Manages reference counting
- Explicitly construct from any C++ object
- Liberal C++ object interoperability

```
def go(x, f):
  # Room for a comment ☺
  if (f == 'foo'):
    x[3:7] = 'bar'
  else:
    x.items += f(3, x)
  return x

def getfunc():
  return go;
```

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

object getfunc()
{
   return object(go);
}
```

Python builtin type wrappers

- list, dict, tuple, str, long,… derived from object
- Act like real Python type: str(1) ⇒ "1"
- Have Python type's methods: d.keys()
- make_tuple for declaring "tuple literals"

```
void f(str name)
{
   object n2 = name.attr("upper")();

   str py_name = name.upper(); // better

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

Derived Object Types

- class_<T> is-a **object**!
- Wraps the Python class object
- Use to create wrapped instances:

```
object v2 =
    class_<Vec2>("Vec2", init<double, double>())
        .def("length", &Point::length)
        .def("angle", &Point::angle);

object vec345 = v2(3.0, 4.0);
assert(vec345.attr("length")() == 5.0);
```

Extracting C++ Objects

- Need to get C++ values out of object instances

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

double l = extract<double>(o.attr("length")());
```

- Need to test extractibility

```
extract<Vec2&> x(o);

if (x.check())
{
    Vec2& v = x();

    ...use v...
```

Iterators

- C++ iterators:
  - 5 type categories (random-access bidirectional forward input output)
  - 2 Operation categories: reposition, access
  - Need a pair to represent a range
- Python Iterators:
  - 1 category (forward)
  - 1 operation category (**next**())
  - Raises StopIteration exception at end

Iterators

Python iteration protocol:

for y in x:

whatever... ≡

```
iter = iter(x)              # calls x.__iter__()
done = false
while not done:
    try:
        y = iter.next()   # get each item
    except StopIteration:
        done = true                 # iterator exhausted
    else:
        whatever                    # process y
```

Iterators – wrapping begin/

- Challenge: produce appropriate __iter__ function

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

- Use in class_<>:

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

Iterators – wrapping any pair

- range(start, finish)
- range<Policies,IterType>(start, finish)
- start/finish may be:
  - member data pointers
  - member function pointers
  - adaptable function object (use IterType param)
- iterator<T, Policies>() – Just calls range with &T::begin, &T::end

Iterators – range +

- Example from LLNL:

```
f = Field()

for x in f.pions:
    smash(x)

for y in f.bogons:
    count(y)
```

- C++ Wrapper:

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

Exception Translation

- C++ exceptions must not propagate into Python!
- Default handler translates selected standard exceptions, then gives up:

  RuntimeError, 'unidentifiable C++ Exception'

- Users may provide custom translation:

```
struct PodBayDoorException;

void translator(PodBayDoorException& x) {
    PyErr_SetString(PyExc_UserWarning, "I'm sorry, Dave…");
}

BOOST_PYTHON_MODULE_INIT(kubrick) {
    register_exception_translator<
        PodBayDoorException>(translator);
    …
```

Pickling (object serialization)

- Python's pickling protocol relies on a subset of three methods:
  - x.__getinitargs__() – get constructor args
  - x.__getstate__() – get additional state
  - x.__setstate__(state) – restore state
- Can define manually, but there are pitfalls:
  - Might define __getstate__ but not __setstate__
  - Might supply wrong signatures
  - Might not handle object's __dict__
- Boost.Python supplies def_pickle / pickle_suite to enforce conformance.

Pyste Examples

```
Class('virtual2::A', 'virtual2.h')
Class('virtual2::B', 'virtual2.h')
Function('virtual2::call', 'virtual2.h')

Point = Template('templates::Point', 'templates.h')
rename(Point.x, 'i')
rename(Point.y, 'j')
IPoint = Point('int')
FPoint = Point('double', 'FPoint')
rename(IPoint, 'IPoint')
rename(IPoint.x, 'x')
rename(IPoint.y, 'y')
```

STAN Template Acceleration

```
template = body[
    table(id="outer", width="100%", height="100%", border="0")[
        tr(valign="bottom")[
            td(id="output", width="75%", valign="top", model="latestOutput")[
                div(pattern="listItem", view="html")[
                    "Foo"
                ]
            ],
            td(id="room-contents", valign="bottom")[
                strong[
                    "Stuff you have"
                ],
                div(model="playerInventory", view="List")[
                    if_(not arg1)[
                        div(_class="item")["Nothing"]
                    ].else_[
                        for_each(arg1)[
                            div(
                                style=["color: red", "color:blue", None]
                              , view="item"
                              , controller="look")[arg1]
                            )
                        ]
                    ]
                ]
            ]
        ]
    ]
]
```

STAN Output

```
<body>
    <table id="outer" width="100%" height="100%" border="0">
        <tr valign="bottom">
            <td id="output" width="75%" valign="top" model="latestOutput">
                <div pattern="listItem" view="html">
                    Foo
                </div>
            </td>
            <td>
                <strong>Stuff you have</strong>
                <div model="playerInventory" view="List">
                    <div class="item">Nothing</div>
                </div>
            </td>
        </tr>
    </table>
</body>
```