Hybrid Development with Boost.Python

Dynamic Language Interoperability

boostpro computing

Agenda

- Learn about Boost.Python
- Promote Hybrid Development

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

boostpro
computing

Deep Differences…

Python vs. C++

Python	C++
Interpreted	Compiled
Dynamic Typing	Statically Typed
Easily Ported	Powerful Compiler
Everything at runtime	Much at compile-time
Flexible	Efficient
Broad library selection	Libs focused on computation

boostpro computing

…But Similar Idioms

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

boostpro computing

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 Binding

- 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)

Let's Expose This to Python

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

   return msgs[x];
}
```

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

Goal

boostpro computing

Hello, Python C API World

```
extern "C"
{
  PyObject* greet_wrap(PyObject* args, PyObject* keywords)
  {
       int x;
       if (PyArg_ParseTuple(args, "i", &x))
           return 0;
       char const* result = greet(x);
       return PyString_FromString(result);
  }

  static PyMethodDef methods[] = {
      { "greet", greet_wrap, METH_VARARGS,
        "Return a greeting" }
    , { NULL, NULL, 0, NULL } // sentinel
  };

  DL_EXPORT init_hello()
  { Py_InitModule("hello", methods); }
}
```

extract/check args

invoke wrapped function

convert back to Python

table of wrapped functions

entry point

High-level Python/C++ binding

- Interface-specification (IDL) ➜ C code (ILU)
- C++-like IDL ➜ C code (SIP)
- Full C++ ➜ C code (SWIG, GRAD)
- C++ wrappers for Python API (CXX)
- Introspective C++ Embedded DSL (Boost)

Hello, Boost.Python World!

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

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

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

boostpro computing

Boost.Python Wrapper EDSL

```
// 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);
}
```

boostpro computing

Function Wrapping Interface

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

- Member/free function duality

```
R (X::*)(A1) ≡ R (*)(X&, A1)
```

- Overloading: use multiple defs with same name

Exposing Classes

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

    std::string greet() const
    { return msg; }

    std::string msg;
};
```

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

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

boostpro computing

Constructors and Bases

```
struct defcon { defcon(); };

struct pair
{
    pair(int, std::string);
};

struct abstract
{
    virtual void f() = 0;
    virtual ~abstract() {}
};

struct derived : pair, defcon
{
    fu(int);
};
```

```
class_<defcon>("defcon");

class_<pair>(
  "pair",
  init<int,string>())

class_<abstract>(
  "abstract",
  no_init() )

class_<derived, bases<pair, defcon> >(
  "derived",
  init<int>() )
```

Object Model

- Exposed classes (and their Python subclasses) contain 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?"

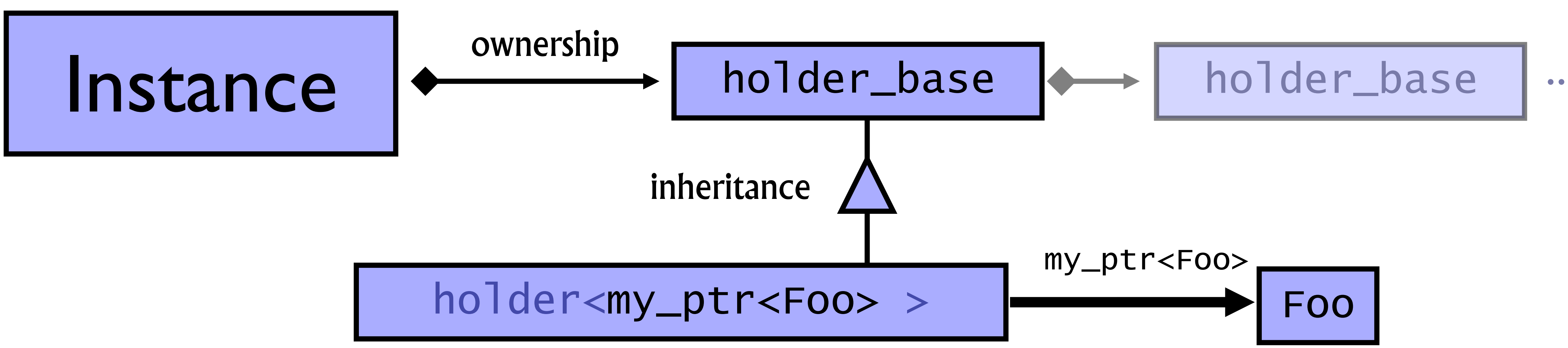
Specifying a holder pointer

```
class Foo { ... };

void f( my_ptr<Foo> );
my_ptr<Foo> g();
```

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

def("f", f);
def("g", g)
```



Class Members

```
struct X
{
    X();
    X(int);

    void foo1();
    void foo2(int);

    char const* val;
}
```

```
class_<X>("X")
.def(init<int>())

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

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

Implied default constructor

Additional constructor

Overload set

Data member

Overridable Virtual Functions

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

int f42(Base& b)
{ return x.f(42); }
```

```
>>> f42( Base() )
42
>>> class D(Base):
...     def f(x):
...         return 2*x
...
>>> f42( D() )
84
```

```
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); }
};

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

def("f42", f42);
```

Exposing Operators

```
class fixed { … };

fixed   operator+(fixed, int);
fixed   operator+(int, fixed);

int     operator-(fixed, fixed);
fixed   operator-(fixed, int);

fixed&  operator+=(fixed&, int);
fixed& operator-=(fixed&, Heavy);

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

```
class_<fixed>()

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

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

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

    .def(self < self)
```

boostpro computing

Other "Special Functions"

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

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

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

```
class_<Num>()

   .def(float_(self))

   .def(pow(self,self))
   .def(abs(self))

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

__float__

__pow__

__abs__

__str__

boostpro computing

Properties

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

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

```
>>> x = Num()
>>> x.value = 3.14
>>> x.value, x.rovalue
(3.14, 3.14)
>>> x.rovalue = 2.17
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

boostpro computing

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?

boostpro computing

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, but...
3.14
```

- Doesn't reflect C++ interface

boostpro computing

Call Policies

- Stored pointers

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

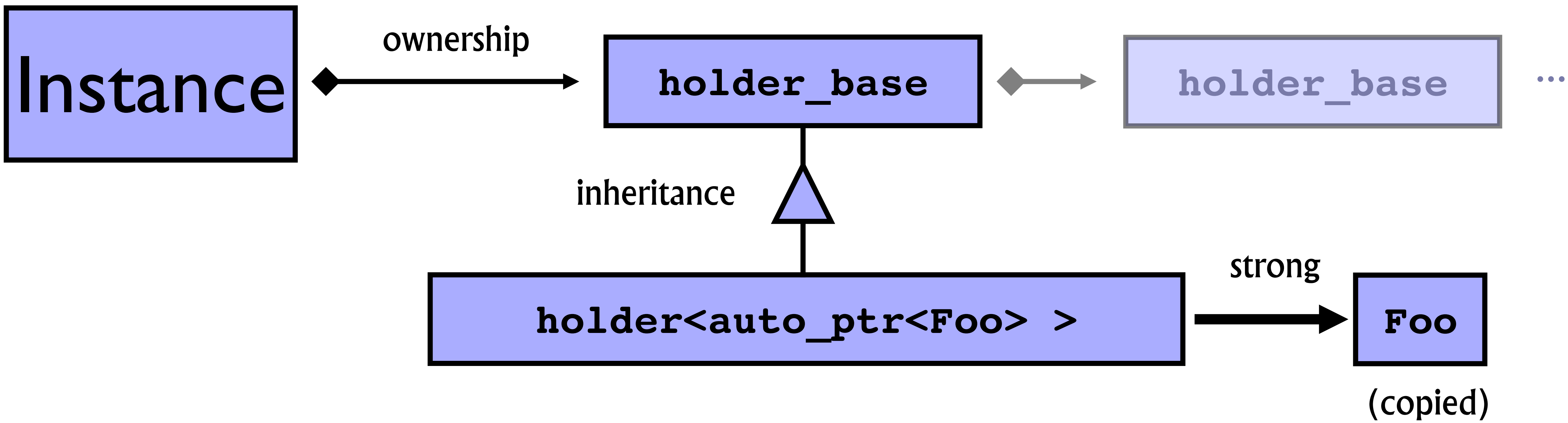
- More problems:

```
>>> 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);
```

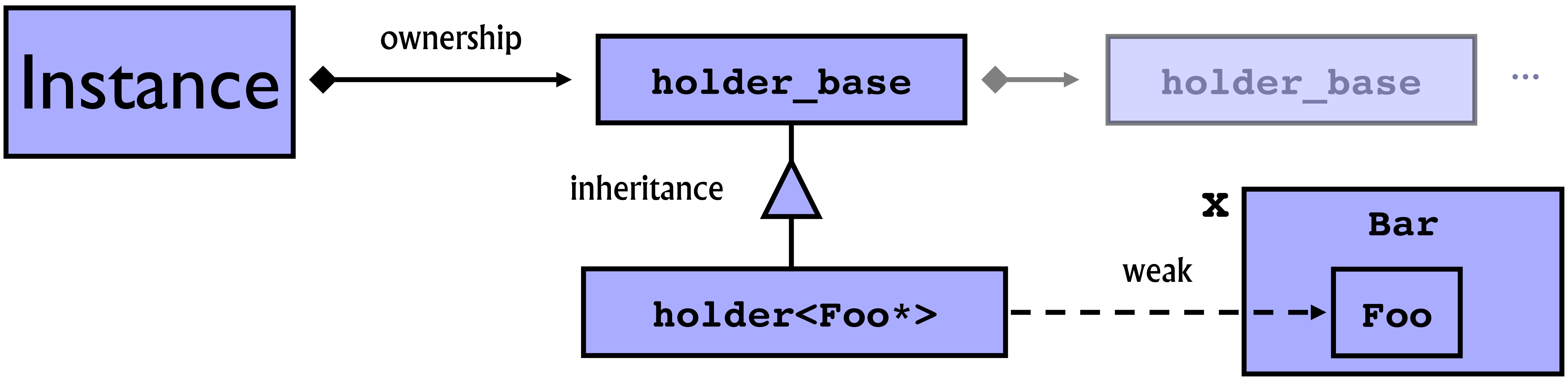


boostpro computing

Wrapped Class Object Model

- Sometimes you want this:

```
Foo const& get(Bar& x);
```



Call Policies - Application

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

```
def(
  "get_x", get_x,
  return_internal_reference<1>() );
```

boostpro computing

Call Policies - Application

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

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

Call Policies - Application

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

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

boostpro computing

Call Policy Chaining

```
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>
    >()
);
```

boostpro computing

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<P>, where P can be:
    - reference_existing_object – naïve (dangerous) approach
    - copy_const_reference – naïve (safe) approach
    - copy_non_const_reference
    - manage_new_object – adopt-a-pointer

Default Arguments

- Boost.Python wraps (member) function pointers
- But 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);
```

boostpro computing

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())
      ;
```

Brief Pause While We Change Reels…

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/end

- 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 + properties

- 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')
```

Py++: http://language-binding.net

```python
#! /usr/bin/python
# Copyright 2004-2008 Roman Yakovenko.
# 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)

import os
import sys
sys.path.append( '../../../..' )
from environment import gccxml
from pyplusplus import module_builder

mb = module_builder.module_builder_t(
        files=['hello_world.hpp']
        , gccxml_path=gccxml.executable ) #path to gccxml executable
```

Exercise

- Expose one of your sparse matrix implementations to Python
- Make it iterable, exposing the stored data
- Expose the row starts and column indices using the pion/bogon technique from slide 50
- Extra credit: make it pickle-able

STAN Template Acceleration Nexus

```
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>
```