Building Hybrid Systems with Boost.Python

Author: David Abrahams
Contact: dave@boost-consulting.com
Organization: Boost Consulting
Date: 2003-03-20
Author: Ralf W. Grosse-Kunstleve
Copyright: Copyright David Abrahams and Ralf W. Grosse-Kunstleve 2003. All rights reserved

Table of Contents

- Abstract
- Introduction
- Boost.Python Design Goals
- Hello Boost.Python World
- Library Overview
  - Exposing Classes
    - Constructors
    - Data Members and Properties
    - Operator Overloading
    - Inheritance
    - Virtual Functions
    - Deeper Reflection on the Horizon?
  - Serialization
  - Object interface
- Thinking hybrid
- Development history
- Conclusions
- Citations
- Footnotes

Abstract

Boost.Python is an open source C++ library which provides a concise IDL-like interface for binding C++ classes and functions to Python. Leveraging the full power of C++ compile-time introspection and of recently developed metaprogramming techniques, this is achieved entirely in pure C++, without introducing a new syntax. Boost.Python's rich set of features and high-level interface make it possible to engineer packages from the ground up as hybrid systems, giving programmers easy and coherent access to both the efficient compile-time polymorphism of C++ and the extremely convenient run-time polymorphism of Python.

Introduction

Python and C++ are in many ways as different as two languages could be: while C++ is usually compiled to machine-code, Python is interpreted. Python's dynamic type system is often cited as the foundation of its flexibility, while in C++ static typing is the cornerstone of its efficiency. C++ has an intricate and difficult compile-time meta-language, while in Python, practically everything happens at runtime.

Yet for many programmers, these very differences mean that Python and C++ complement one another perfectly. Performance bottlenecks in Python programs can be rewritten in C++ for maximal speed, and authors of powerful C++ libraries choose Python as a middleware language for its flexible system integration capabilities. Furthermore, the surface differences mask some strong similarities:

- 'C'-family control structures (if, while, for...)
- Support for object-orientation, functional programming, and generic programming (these are both *multi-paradigm* programming languages.)
- Comprehensive operator overloading facilities, recognizing the importance of syntactic variability for readability and expressivity.
- High-level concepts such as collections and iterators.
- High-level encapsulation facilities (C++: namespaces, Python: modules) to support the design of re-usable libraries.
- Exception-handling for effective management of error conditions.
- C++ idioms in common use, such as handle/body classes and reference-counted smart pointers mirror Python reference semantics.

Given Python's rich 'C' interoperability API, it should in principle be possible to expose C++ type and function interfaces to Python with an analogous interface to their C++ counterparts. However, the facilities provided by Python alone for integration with C++ are relatively meager. Compared to C++ and Python, 'C' has only very rudimentary abstraction facilities, and support for exception-handling is completely missing. 'C' extension module writers are required to manually manage Python reference counts, which is both annoyingly tedious and extremely error-prone. Traditional extension modules also tend to contain a great deal of boilerplate code repetition which makes them difficult to maintain, especially when wrapping an evolving API.

These limitations have lead to the development of a variety of wrapping systems. SWIG is probably the most popular package for the integration of C/C++ and Python. A more recent development is SIP, which was specifically designed for interfacing Python with the Qt graphical user interface library. Both SWIG and SIP introduce their own specialized languages for customizing inter-language bindings. This has certain advantages, but having to deal with three different languages (Python, C/C++ and the interface language) also introduces practical and mental difficulties. The CXX package demonstrates an interesting alternative. It shows that at least some parts of Python's 'C' API can be wrapped and presented through a much more user-friendly C++ interface. However, unlike SWIG and SIP, CXX does not include support for wrapping C++ classes as new Python types.

The features and goals of Boost.Python overlap significantly with many of these other systems. That said, Boost.Python attempts to maximize convenience and flexibility without introducing a separate wrapping language. Instead, it presents the user with a high-level C++ interface for wrapping C++ classes and functions, managing much of the complexity behind-the-scenes with static metaprogramming. Boost.Python also goes beyond the scope of earlier systems by providing:

- Support for C++ virtual functions that can be overridden in Python.
- Comprehensive lifetime management facilities for low-level C++ pointers and references.
- Support for organizing extensions as Python packages, with a central registry for inter-language type conversions.
- A safe and convenient mechanism for tying into Python's powerful serialization engine (pickle).
- Coherence with the rules for handling C++ lvalues and rvalues that can only come from a deep understanding of both the Python and C++ type systems.

The key insight that sparked the development of Boost.Python is that much of the boilerplate code in traditional extension modules could be eliminated using C++ compile-time introspection. Each argument of a wrapped C++ function must be extracted from a Python object using a procedure that depends on the argument type. Similarly the function's return type determines how the return value will be converted from C++ to Python. Of course argument and return types are part of each function's type, and this is exactly the source from which Boost.Python deduces most of the information required.

This approach leads to *user guided wrapping*: as much information is extracted directly from the source code to be wrapped as is possible within the framework of pure C++, and some additional information is supplied explicitly by the user. Mostly the guidance is mechanical and little real intervention is required. Because the interface specification is written in the same full-featured language as the code being exposed, the user has unprecedented power available when she does need to take control.

Boost.Python Design Goals

The primary goal of Boost.Python is to allow users to expose C++ classes and functions to Python using nothing more than a C++ compiler. In broad strokes, the user experience should be one of directly manipulating C++ objects from Python.

However, it's also important not to translate all interfaces *too* literally: the idioms of each language must be respected. For example, though C++ and Python both have an iterator concept, they are expressed very differently. Boost.Python has to be able to bridge the interface gap.

It must be possible to insulate Python users from crashes resulting from trivial misuses of C++ interfaces, such as accessing already-deleted objects. By the same token the library should insulate C++ users from low-level Python 'C' API, replacing error-prone 'C' interfaces like manual reference-count management and raw `PyObject` pointers with more-robust alternatives.

Support for component-based development is crucial, so that C++ types exposed in one extension module can be passed to functions exposed in another without loss of crucial information like C++ inheritance relationships.

Finally, all wrapping must be *non-intrusive*, without modifying or even seeing the original C++ source code. Existing C++ libraries have to be wrappable by third parties who only have access to header files and binaries.

Hello Boost.Python World

And now for a preview of Boost.Python, and how it improves on the raw facilities offered by Python. Here's a function we might want to expose:

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

   if (x > 2)
       throw std::range_error(``greet: index out of range'');

   return msgs[x];
}
```

To wrap this function in standard C++ using the Python 'C' API, we'd need something like this:

```
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 result 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()
    {
        (void) Py_InitModule(``hello'', methods); // add the methods to the module
    }
}
```

Now here's the wrapping code we'd use to expose it with Boost.Python:

```
#include <boost/python.hpp>
using namespace boost::python;
BOOST_PYTHON_MODULE(hello)
{
    def(``greet'', greet, ``return one of 3 parts of a greeting'');
}
```

and here it is in action:

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

Aside from the fact that the 'C' API version is much more verbose, it's worth noting a few things that it doesn't handle correctly:

- The original function accepts an unsigned integer, and the Python 'C' API only gives us a way of extracting signed integers. The Boost.Python version will raise a Python exception if we try to pass a negative number to `hello.greet`, but the other one will proceed to do whatever the C++ implementation does when converting an negative integer to unsigned (usually wrapping to some very large number), and pass the incorrect translation on to the wrapped function.
- That brings us to the second problem: if the C++ `greet()` function is called with a number greater than 2, it will throw an exception. Typically, if a C++ exception propagates across the boundary with code generated by a 'C' compiler, it will cause a crash. As you can se(f) -341 3ouit vt,here's.963no5 (s.963C 1 (++) -27cafthro) fold(erting) -2t50 (h

Exposing Classes

C++ classes and structs are exposed with a similarly-terse interface. Given:

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

The following code will expose it in our extension module:

```
#include <boost/python.hpp>
BOOST_PYTHON_MODULE(hello)
{
    class_<World>(``World'')
        .def(``greet'', &World::greet)
        .def(``set'', &World::set)
    ;
}
```

Although this code has a certain pythonic familiarity, people sometimes find the syntax bit confusing because it doesn't look like most of the C++ code they're used to. All the same, this is just standard C++. Because of their flexible syntax and operator overloading, C++ and Python are great for defining domain-specific (sub)languages (DSLs), and that's what we've done in Boost.Python. To break it down:

```
class_<World>(``World'')
```

constructs an unnamed object of type `class_<World>` and passes `` `World' `` to its constructor. This creates a new-style Python class called `World` in the extension module, and associates it with the C++ type `World` in the Boost.Python type conversion registry. We might have also written:

```
class_<World> w(``World'');
```

but that would've been more verbose, since we'd have to name `w` again to invoke its `def()` member function:

```
w.def(``greet'', &World::greet)
```

There's nothing special about the location of the dot for member access in the original example: C++ allows any amount of whitespace on either side of a token, and placing the dot at the beginning of each line allows us to chain as many successive calls to member functions as we like with a uniform syntax. The other key fact that allows chaining is that `class_<>` member functions all return a reference to `*this`.

So the example is equivalent to:

```
class_<World> w(``World'');
w.def(``greet'', &World::greet);
w.def(``set'', &World::set);
```

It's occasionally useful to be able to break down the components of a Boost.Python class wrapper in this way, but the rest of this article will stick to the terse syntax.

For completeness, here's the wrapped class in use:

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

Constructors

Since our `World` class is just a plain `struct`, it has an implicit no-argument (nullary) constructor. Boost.Python exposes the nullary constructor by default, which is why we were able to write:

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

However, well-designed classes in any language may require constructor arguments in order to establish their invariants. Unlike Python, where `__init__` is just a specially-named method, In C++ constructors cannot be handled like ordinary member functions. In particular, we can't take their address: `&World::World` is an error. The library provides a different interface for specifying constructors. Given:

```
struct World
{
    World(std::string msg); // added constructor
    ...
```

we can modify our wrapping code as follows:

```
class_<World>(''World'', init<std::string>())
    ...
```

of course, a C++ class may have additional constructors, and we can expose those as well by passing more instances of `init<...>` to `def()`:

```
class_<World>(''World'', init<std::string>())
    .def(init<double, double>())
    ...
```

Boost.Python allows wrapped functions, member functions, and constructors to be overloaded to mirror C++ overloading.

Data Members and Properties

Any publicly-accessible data members in a C++ class can be easily exposed as either `readonly` or `readwrite` attributes:

```
class_<World>(''World'', init<std::string>())
    .def_readonly(''msg'', &World::msg)
    ...
```

and can be used directly in Python:

```
>>> planet = hello.World('howdy')
>>> planet.msg
'howdy'
```

This does *not* result in adding attributes to the `World` instance `__dict__`, which can result in substantial memory savings when wrapping large data structures. In fact, no instance `__dict__` will be created at all unless attributes are explicitly added from Python. Boost.Python owes this capability to the new Python 2.2 type system, in particular the descriptor interface and `property` type.

In C++, publicly-accessible data members are considered a sign of poor design because they break encapsulation, and style guides usually dictate the use of "getter" and "setter" functions instead. In Python, however, `__getattr__`, `__setattr__`, and since 2.2, `property` mean that attribute access is just one more well-encapsulated syntactic tool at the programmer's disposal. Boost.Python bridges this idiomatic gap by making Python `property` creation directly available to users. If `msg` were private, we could still expose it as attribute in Python as follows:

```
class_<World>(''World'', init<std::string>())
    .add_property(''msg'', &World::greet, &World::set)
    ...
```

The example above mirrors the familiar usage of properties in Python 2.2+:

```
>>> class World(object):
...     __init__(self, msg):
...         self.__msg = msg
...     def greet(self):
...         return self.__msg
...     def set(self, msg):
...         self.__msg = msg
...     msg = property(greet, set)
```

Operator Overloading

The ability to write arithmetic operators for user-defined types has been a major factor in the success of both languages for numerical computation, and the success of packages like NumPy attests to the power of exposing operators in extension modules. Boost.Python provides a concise mechanism for wrapping operator overloads. The example below shows a fragment from a wrapper for the Boost rational number library:

```
class_<rational<int> >(``rational_int'')
  .def(init<int, int>()) // constructor, e.g. rational_int(3,4)
  .def(``numerator'', &rational<int>::numerator)
  .def(``denominator'', &rational<int>::denominator)
  .def(-self)         // __neg__ (unary minus)
  .def(self + self)   // __add__ (homogeneous)
  .def(self * self)   // __mul__
  .def(self + int()) // __add__ (heterogenous)
  .def(int() + self) // __radd__
  ...
```

The magic is performed using a simplified application of "expression templates" [VELD1995], a technique originally developed for optimization of high-performance matrix algebra expressions. The essence is that instead of performing the computation immediately, operators are overloaded to construct a type *representing* the computation. In matrix algebra, dramatic optimizations are often available when the structure of an entire expression can be taken into account, rather than evaluating each operation "greedily". Boost.Python uses the same technique to build an appropriate Python method object based on expressions involving `self`.

Inheritance

C++ inheritance relationships can be represented to Boost.Python by adding an optional `bases<...>` argument to the `class_<...>` template parameter list as follows:

```
class_<Derived, bases<Base1,Base2> >(``Derived'')
     ...
```

This has two effects:

1. When the `class_<...>` is created, Python type objects corresponding to `Base1` and `Base2` are looked up in Boost.Python's registry, and are used as bases for the new Python `Derived` type object, so methods exposed for the Python `Base1` and `Base2` types are automatically members of the `Derived` type. Because the registry is global, this works correctly even if `Derived` is exposed in a different module from either of its bases.

2. C++ conversions from `Derived` to its bases are added to the Boost.Python registry. Thus wrapped C++ methods expecting (a pointer or reference to) an object of either base type can be called with an object wrapping a `Derived` instance. Wrapped member functions of class `T` are treated as though they have an implicit first argument of `T&`, so these conversions are neccessary to allow the base class methods to be called for derived objects.

Of course it's possible to derive new Python classes from wrapped C++ class instances. Because Boost.Python uses the new-style class system, that works very much as for the Python built-in types. There is one significant detail in which it differs: the built-in types generally establish their invariants in their `__new__` function, so that derived classes do not need to call `__init__` on the base class before invoking its methods :

```
>>> class L(list):
...     def __init__(self):
...         pass
...
>>> L().reverse()
>>>
```

Because C++ object construction is a one-step operation, C++ instance data cannot be constructed until the arguments are available, in the `__init__` function:

```
>>> class D(SomeBoostPythonClass):
...     def __init__(self):
...         pass
...
>>> D().some_boost_python_method()
Traceback (most recent call last):
  File ``<stdin>'', line 1, in ?
TypeError: bad argument type for built-in operation
```

This happened because Boost.Python couldn't find instance data of type `SomeBoostPythonClass` within the `D` instance; `D`'s `__init__` function masked construction of the base class. It could be corrected by either removing `D`'s `__init__` function or having it call `SomeBoostPythonClass.__init__(...)` explicitly.

Virtual Functions

Deriving new types in Python from extension classes is not very interesting unless they can be used polymorphically from C++. In other words, Python method implementations should appear to override the implementation of C++ virtual functions when called *through base class pointers/references from C++*. Since the only way to alter the behavior of a virtual function is to override it in a derived class, the user must build a special derived class to dispatch a polymorphic class' virtual functions:

```
//
// interface to wrap:
//
class Base
{
 public:
    virtual int f(std::string x) { return 42; }
    virtual ~ Base();
};

int calls_f(Base const& b, std::string x) { return b.f(x); }

//
// Wrapping Code
//

// Dispatcher class
struct BaseWrap : Base
{
    // Store a pointer to the Python object
    BaseWrap(PyObject* self_) : self(self_) {}
    PyObject* self;

    // Default implementation, for when f is not overridden
    int f_default(std::string x) { return this->Base::f(x); }
    // Dispatch implementation
    int f(std::string x) { return call_method<int>(self, ``f'', x); }
};
```

```
...
    def(``calls_f'', calls_f);
    class_<Base, BaseWrap>(``Base'')
        .def(``f'', &Base::f, &BaseWrap::f_default)
        ;
```

Now here's some Python code which demonstrates:

```
>>> class Derived(Base):
...     def f(self, s):
...          return len(s)
...
>>> calls_f(Base(), 'foo')
42
>>> calls_f(Derived(), 'forty-two')
9
```

Things to notice about the dispatcher class:

- The key element which allows overriding in Python is the `call_method` invocation, which uses the same global type conversion registry as the C++ function wrapping does to convert its arguments from C++ to Python and its return type from Python to C++.
- Any constructor signatures you wish to wrap must be replicated with an initial `PyObject*` argument
- The dispatcher must store this argument so that it can be used to invoke `call_method`
- The `f_default` member function is needed when the function being exposed is not pure virtual; there's no other way `Base::f` can be called on an object of type `BaseWrap`, since it overrides `f`.

Deeper Reflection on the Horizon?

Admittedly, this formula is tedious to repeat, especially on a project with many polymorphic classes. That it is neccessary reflects some limitations in C++'s compile-time introspection capabilities: there's no way to enumerate the members of a class and find out which are virtual functions. At least one very promising project has been started to write a front-end which can generate these dispatchers (and other wrapping code) automatically from C++ headers.

Pyste is being developed by Bruno da Silva de Oliveira. It builds on GCC_XML, which generates an XML version of GCC's internal program representation. Since GCC is a highly-conformant C++ compiler, this ensures correct handling of the most-sophisticated template code and full access to the underlying type system. In keeping with the Boost.Python philosophy, a Pyste interface description is neither intrusive on the code being wrapped, nor expressed in some unfamiliar language: instead it is a 100% pure Python script. If Pyste is successful it will mark a move away from wrapping everything directly in C++ for many of our users. It will also allow us the choice to shift some of the metaprogram code from C++ to Python. We expect that soon, not only our users but the Boost.Python developers themselves will be "thinking hybrid" about their own code.

Serialization

Serialization is the process of converting objects in memory to a form that can be stored on disk or sent over a network connection. The serialized object (most often a plain string) can be retrieved and converted back to the original object. A good serialization system will automatically convert entire object hierarchies. Python's standard `pickle` module is just such a system. It leverages the language's strong runtime introspection facilities for serializing practically arbitrary user-defined objects. With a few simple and unintrusive provisions this powerful machinery can be extended to also work for wrapped C++ objects. Here is an example:

```
#include <string>

struct World
{
    World(std::string a_msg) : msg(a_msg) {}
    std::string greet() const { return msg; }
```

```
    std::string msg;
};

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

struct World_picklers : pickle_suite
{
  static tuple
  getinitargs(World const& w) { return make_tuple(w.greet()); }
};

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

Now let's create a `World` object and put it to rest on disk:

```
>>> import hello
>>> import pickle
>>> a_world = hello.World(``howdy'')
>>> pickle.dump(a_world, open(``my_world'', ``w''))
```

In a potentially *different script* on a potentially *different computer* with a potentially *different operating system*:

```
>>> import pickle
>>> resurrected_world = pickle.load(open(``my_world'', ``r''))
>>> resurrected_world.greet()
'howdy'
```

Of course the `cPickle` module can also be used for faster processing.

Boost.Python's `pickle_suite` fully supports the `pickle` protocol defined in the standard Python documentation. Like a __getinitargs__ function in Python, the pickle_suite's getinitargs() is responsible for creating the argument tuple that will be use to reconstruct the pickled object. The other elements of the Python pickling protocol, __getstate__ and __setstate__ can be optionally provided via C++ getstate and setstate functions. C++'s static type system allows the library to ensure at compile-time that nonsensical combinations of functions (e.g. getstate without setstate) are not used.

Enabling serialization of more complex C++ objects requires a little more work than is shown in the example above. Fortunately the `object` interface (see next section) greatly helps in keeping the code manageable.

Object interface

Experienced 'C' language extension module authors will be familiar with the ubiquitous `PyObject*`, manual reference-counting, and the need to remember which API calls return "new" (owned) references or "borrowed" (raw) references. These constraints are not just cumbersome but also a major source of errors, especially in the presence of exceptions.

Boost.Python provides a class `object` which automates reference counting and provides conversion to Python from C++ objects of arbitrary type. This significantly reduces the learning effort for prospective extension module writers.

Creating an `object` from any other type is extremely simple:

```
object s(``hello, world'');  // s manages a Python string
```

`object` has templated interactions with all other types, with automatic to-python conversions. It happens so naturally that it's easily overlooked:

```
object ten_Os = 10 * s[4]; // -> ``oooooooooo''
```

In the example above, `4` and `10` are converted to Python objects before the indexing and multiplication operations are invoked.

The `extract<T>` class template can be used to convert Python objects to C++ types:

```
double x = extract<double>(o);
```

If a conversion in either direction cannot be performed, an appropriate exception is thrown at runtime.

The `object` type is accompanied by a set of derived types that mirror the Python built-in types such as `list`, `dict`, `tuple`, etc. as much as possible. This enables convenient manipulation of these high-level types from C++:

```
dict d;
d[``some''] = ``thing'';
d[``lucky_number''] = 13;
list l = d.keys();
```

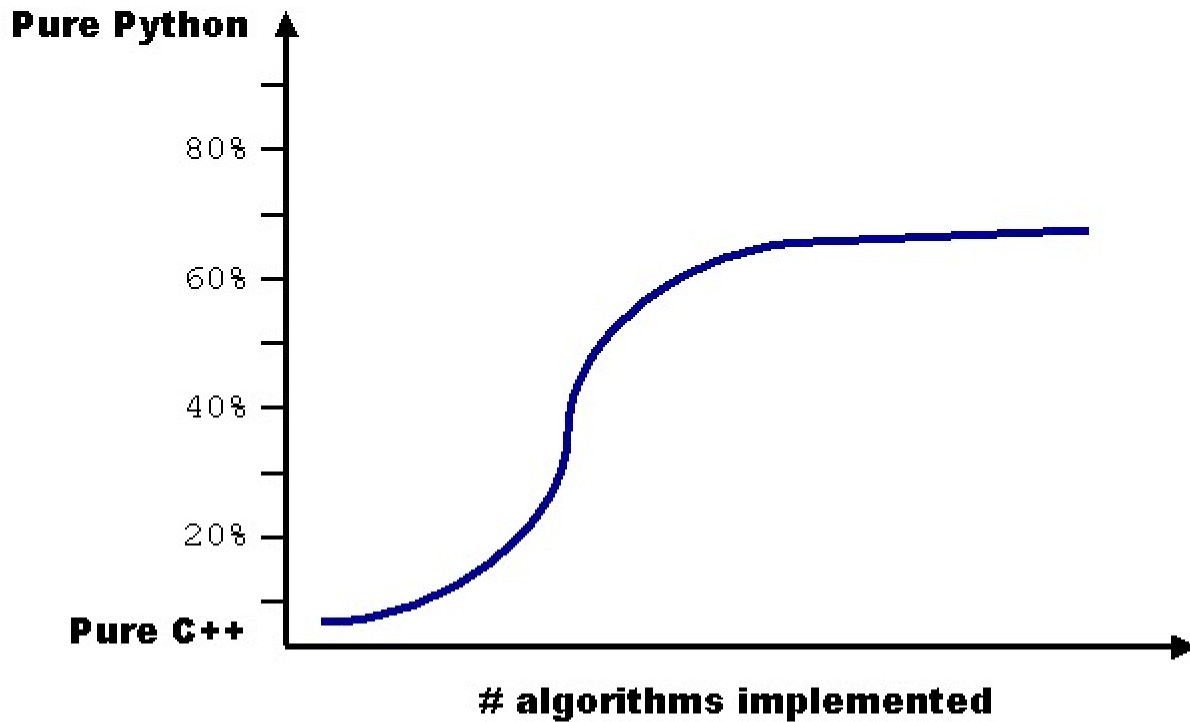
This almost looks and works like regular Python code, but it is pure C++. Of course we can wrap C++ functions which accept or return `object` instances.

Thinking hybrid

Because of the practical and mental difficulties of combining programming languages, it is common to settle a single language at the outset of any development effort. For many applications, performance considerations dictate the use of a compiled language for the core algorithms. Unfortunately, due to the complexity of the static type system, the price we pay for runtime performance is often a significant increase in development time. Experience shows that writing maintainable C++ code usually takes longer and requires *far* more hard-earned working experience than developing comparable Python code. Even when developers are comfortable working exclusively in compiled languages, they often augment their systems by some type of ad hoc scripting layer for the benefit of their users without ever availing themselves of the same advantages.

Boost.Python enables us to *think hybrid*. Python can be used for rapidly prototyping a new application; its ease of use and the large pool of standard libraries give us a head start on the way to a working system. If necessary, the working code can be used to discover rate-limiting hotspots. To maximize performance these can be reimplemented in C++, together with the Boost.Python bindings needed to tie them back into the existing higher-level procedure.

Of course, this *top-down* approach is less attractive if it is clear from the start that many algorithms will eventually have to be implemented in C++. Fortunately Boost.Python also enables us to pursue a *bottom-up* approach. We have used this approach very successfully in the development of a toolbox for scientific applications. The toolbox started out mainly as a library of C++ classes with Boost.Python bindings, and for a while the growth was mainly concentrated on the C++ parts. However, as the toolbox is becoming more complete, more and more newly added functionality can be implemented in Python.



This figure shows the estimated ratio of newly added C++ and Python code over time as new algorithms are implemented. We expect this ratio to level out near 70% Python. Being able to solve new problems mostly in Python rather than a more difficult statically typed language is the return on our investment in Boost.Python. The ability to access all of our code from Python allows a broader group of developers to use it in the rapid development of new applications.

Development history

The first version of Boost.Python was developed in 2000 by Dave Abrahams at Dragon Systems, where he was privileged to have Tim Peters as a guide to "The Zen of Python". One of Dave's jobs was to develop a Python-based natural language processing system. Since it was eventually going to be targeting embedded hardware, it was always assumed that the compute-intensive core would be rewritten in C++ to optimize speed and memory footprint [1]. The project also wanted to test all of its C++ code using Python test scripts [2]. The only tool we knew of for binding C++ and Python was SWIG, and at the time its handling of C++ was weak. It would be false to claim any deep insight into the possible advantages of Boost.Python's approach at this point. Dave's interest and expertise in fancy C++ template tricks had just reached the point where he could do some real damage, and Boost.Python emerged as it did because it filled a need and because it seemed like a cool thing to try.

This early version was aimed at many of the same basic goals we've described in this paper, differing most-noticeably by having a slightly more cumbersome syntax and by lack of special support for operator overloading, pickling, and component-based development. These last three features were quickly added by Ullrich Koethe and Ralf Grosse-Kunstleve [3], and other enthusiastic contributors arrived on the scene to contribute enhancements like support for nested modules and static member functions.

By early 2001 development had stabilized and few new features were being added, however a disturbing new fact came to light: Ralf had begun testing Boost.Python on pre-release versions of a compiler using the EDG front-end, and the mechanism at the core of Boost.Python responsible for handling conversions between Python and C++ types was failing to compile. As it turned out, we had been exploiting a very common bug in the implementation of all the C++ compilers we had tested. We knew that as C++ compilers rapidly became more standards-compliant, the library would begin failing on more platforms. Unfortunately, because the mechanism was so central to the functioning of the library, fixing the problem looked very difficult.

Fortunately, later that year Lawrence Berkeley and later Lawrence Livermore National labs contracted with Boost Consulting for support and development of Boost.Python, and there was a new opportunity to address fundamental issues and ensure a future for the library. A redesign effort began with the low level type conversion architecture, building in standards-compliance and support for component-based development (in contrast to version 1 where conversions had to be explicitly imported and exported across module boundaries). A new analysis of the relationship between the Python and C++ objects was done, resulting in more intuitive handling for C++ lvalues and rvalues.

The emergence of a powerful new type system in Python 2.2 made the choice of whether to maintain compatibility with Python 1.5.2 easy: the opportunity to throw away a great deal of elaborate code for emulating classic Python classes alone was too good to pass up. In addition, Python iterators and descriptors provided crucial and elegant tools for representing similar C++ constructs. The development of the generalized `object` interface allowed us to further shield C++ programmers from the dangers and syntactic burdens of the Python 'C' API. A great number of other features including C++ exception translation, improved support for overloaded functions, and most significantly, CallPolicies for handling pointers and references, were added during this period.

In October 2002, version 2 of Boost.Python was released. Development since then has concentrated on improved support for C++ runtime polymorphism and smart pointers. Peter Dimov's ingenious `boost::shared_ptr` design in particular has allowed us to give the hybrid developer a consistent interface for moving objects back and forth across the language barrier without loss of information. At first, we were concerned that the sophistication and complexity of the Boost.Python v2 implementation might discourage contributors, but the emergence of Pyste and several other significant feature contributions have laid those fears to rest. Daily questions on the Python C++-sig and a backlog of desired improvements show that the library is getting used. To us, the future looks bright.

Conclusions

Boost.Python achieves seamless interoperability between two rich and complimentary language environments. Because it leverages template metaprogramming to introspect about types and functions, the user never has to learn a third syntax: the interface definitions are written in concise and maintainable C++. Also, the wrapping system doesn't have to parse C++ headers or represent the type system: the compiler does that work for us.

Computationally intensive tasks play to the strengths of C++ and are often impossible to implement efficiently in pure Python, while jobs like serialization that are trivial in Python can be very difficult in pure C++. Given the luxury of building a hybrid software system from the ground up, we can approach design with new confidence and power.

Citations

Footnotes

[VELD1995] T. Veldhuizen, "Expression Templates," C++ Report, Vol. 7 No. 5 June 1995, pp. 26-31. http://osl.iu.edu/~tveldhui/papers/Expression-Templates/exprtmpl.html

[1] In retrospect, it seems that "thinking hybrid" from the ground up might have been better for the NLP system: the natural component boundaries defined by the pure python prototype turned out to be inappropriate for getting the desired performance and memory footprint out of the C++ core, which eventually caused some redesign overhead on the Python side when the core was moved to C++.

[2] We also have some reservations about driving all C++ testing through a Python interface, unless that's the only way it will be ultimately used. Any transition across language boundaries with such different object models can inevitably mask bugs.

[3] These features were expressed very differently in v1 of Boost.Python