Bridging Python to C++ - and vice-versa

Jack Jansen
Centrum voor Wiskunde en Informatica
Jack.Jansen@cwi.nl

Abstract

I found myself in need of good access to C++ functionality from Python. By *good* I mean: more than simply being able to call C++ routines or methods, but also the ability to subclass C++ baseclasses in Python, and subclass Python baseclasses in C++, similar to the functionality *PyObjC [1]* provides when you need interoperability between Python and Objective-C.

The objective is tackled by extending the standard (but little known:-) *bgen* bridge to be able to parse C++ headers, teaching it about C++ objects, inheritance and callbacks and exposing the result as a Python extension module.

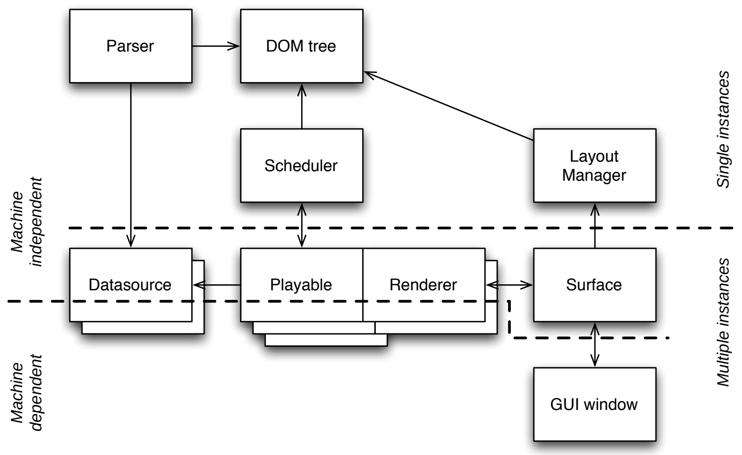
There are various other solutions that allow you to automatically generate Python interfaces to C++ libraries, but I could not find any that I really liked: most are lacking in functionality, such as bidirectional subclassing, some are bloatware, some are ugly. So in good open source tradition I decided to try and do better.

At the time of writing (October 2005) this is still work in progress, so this paper will focus more on the solutions chosen than on the actual tool.

The problem

Over the last two years the SEN5 group at CWI has been busy designing and implementing the Ambulant Player *[2]*, a multimedia playback engine. One of the design goals for the Ambulant Player is to provide the research community (including ourselves) with a framework multimedia player in which all components can easily be replaced to test new ideas, while all other components continue working. This allows one to concentrate on the issue at hand, such as new scheduling algorithms, without having to worry about implementing communication, rendering, user interfaces, etc. For the purpose of this paper we will mention only one other design goal: the implementation should be CPU-efficient and portable, so it can be deployed not only on well-connected desktop machines but also on low-power handheld devices with relatively low-bandwidth network connections.

The figure shows the general design of the playback engine, with multiple stacked boxes denoting that there may be more than one implementation of a module (renderers for different media types, for example). Each of the components is replaceable, and all the interfaces are well-defined.



Because of the efficiency goal C++ was chosen as the initial implementation language, even though it somewhat conflicts with the goal of easy extensibility. Previous experience of the team with using the Python-based *GRiNS* player on low-power devices were rather disappointing, and similar stories about Java and other HLLs seemed to corroborate that.

This summer we started on an attempt to have our cake and eat it too, by opening up the Ambulant API to Python. This will allow rapid development of new ideas, while the performance cost of using a HLL is only incurred in that case. Because we wanted each and any component to be replaceable we needed a bridging solution that not only allowed Python code to call functions and methods in C++, but also wrapping Python objects transparently as C++ objects and subclassing of C++ baseclasses in Python and vice versa.

Solutions we did not pick

The first solution we did not pick was a hand-coded bridge. The current API consists of about 50 interfaces with about 250 methods total, and as the engine is still in active development it is a moving target. This would make the maintenance cost of hand coding, even with tools such as Pyrex, too high.

So we needed an automatic tool to generate the bridge, which quickly pointed to the Bgen tool because the author happens to be pretty familiar with it. But, of course, in a paper one should state that various solutions were examined and found wanting, so here is a list of tools we did not use:

- Swig *[3]* is probably the most popular tool, and has the advantage that it can generate bridges to various languages. But it has a very serious drawback: the code it generates is a mix of C or C++ and Python, and pretty much impossible to debug, even attempting to read it is a challenge. Support for additional datatypes is also a pain and requires adding directives to the `.h` or `.i` files. And new, incompatible versions are released every couple of months.
- Boost *[4]* comes with a huge embedding framework that appears to be impossible to work around. While low-power devices are not a primary target for the bridge this still appeared to be a bit too much of a good thing.
- In all fairness, Sip *[5]* was not on the radar until too late (about two weeks before this paper was written:-). It seems to be more stable and produce better code than Swig, but it shares its directive problem.

The C++ and Python object model

I will assume that readers are familiar with the C++ object model, and only step aside for a moment to explain the main points about the Python object model. We concentrate here on the so-called "new style classes", which were introduced in Python 2.2. The definitive reference for these are PEP 252 *[6]* and PEP 253 *[7]*.

Here I will only say that the new Python object model provides enough framework to interoperate with all the behavior that other languages may require: splitting of allocation and initialization of objects, easy calling of base class methods, cross-language access to members in base classes and much more.

The bgen tool

Bgen is a little-known part of the Python core distribution. It was written over 10 years ago by Guido van Rossum, and while it is in principle a general tool its main application has always been the generation of Python extension modules that interface to the MacOS toolbox modules. This history is reflected in a number of design choices, but the generality of the tool has allowed it to keep up with the changing times fairly easily.

Here is what bgen does in a nutshell:

1. Parse a standard C ".h" file, without any adornments such Swig or Sip require. This is done with a regular expression parser that has very limited knowledge of the C language, but manages to find the function declarations and deduce the parameters, etc.

```
EXTERN_API( void )
HideDialogItem(
      DialogRef         theDialog,
      DialogItemIndex   itemNo)                      ONEWORDINLINE(0xA827);
```

```
EXTERN_API( Boolean )
IsDialogEvent(const EventRecord * theEvent)        ONEWORDINLINE(0xA97F);
```

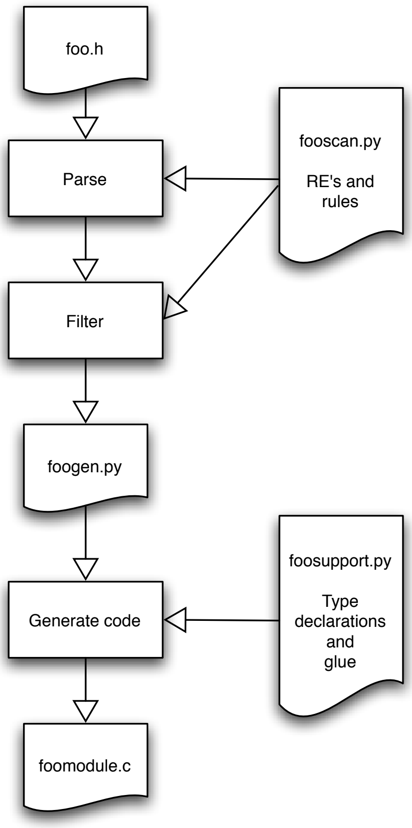
2. Read a user-supplied set of filters and transforms, and apply these to the declarations. These transforms can change parameter signatures in a pretty powerful way. This is important, because it allows things like representing a C negative return value as an exception, or a pair of arguments "`const char *buf, int *bufsize`" as a Python string. The filter can also drop declarations based on unsupported types, names, etc.

3. Decide whether the resulting declaration should be treated as a normal function or as a method. Many C toolkits follow a somewhat object-oriented paradigm by having a standard first (or last) argument, as in `HideDialogItem(DialogRef theDialog, ...)`. These will become methods of the Window object on the Python side without any additional glue code such as Swig needs.

4. The resulting declarations are dumped as Python source, which will be executed later.

```
f = Method(void, 'HideDialogItem',
    (DialogRef, 'theDialog', InMode),
    (DialogItemIndex, 'itemNo', InMode),
)
methods.append(f)

f = Function(Boolean, 'IsDialogEvent',
    (EventRecord_ptr, 'theEvent', InMode),
)
functions.append(f)
```



5. Read a user-supplied file of type declarations. Bgen contains an extensible library of predefined types, for which it knows how these should be initialized, converted from Python to C and vice-versa, passed to C and cleaned up. The library not only contains the standard ints, floats and strings but also various types of buffers and exception/error conditions, and wrapper types that encapsulate a C type, usually a pointer, in a Python object. The extensible nature is important here, because together with the transformation mentioned earlier it allows fairly natural signatures on the Python side.

```
DialogRef = OpaqueByValueType("DialogRef", "DlgObj")
DialogItemIndex = Type("DialogItemIndex", "h")
```

6. Now it is time to read the function and method declarations again. Usually the type declarations will include at least one wrapped object to which the methods are added. The non-method functions are added to a module object, as are the wrapper objects.

```
module = MacModule('_Dlg', 'Dlg', includestuff, finalstuff, initstuff)
object = MyObjectDefinition('Dialog', 'DlgObj', 'DialogRef')
module.addobject(object)

execfile("dlggen.py")

for f in functions: module.add(f)
for f in methods: object.add(f)
```

7. Finally, bgen recursively iterates over the module to generate the C code. This is actually a fairly simple process, because a lot of delegation is used. For example, to generate the argument parsing a method generator will first call the "declare" and "initialize" method of each argument (which will in turn delegate to their types), then output the first bit of the `PyArg_Parse` call, then call the `getArgsFormat` method for each argument to return the format character(s), then call `getArgsArgs` for each argument to get the arguments needed for the `PyArg_Parse` call.

Extending bgen to C++

Extending Bgen to work for C++ in stead of for C required work in a number of areas. First of all, the parser needs to become quite a bit more powerful, because it will have to learn about scopes (for classes and namespaces) plus all the extra ~~gunk~~ functionality such as argument default values and inline functions. Initially we thought a tool like gcc-xml *[8]* would be needed to parse the header files, but luckily we decided to try a quick regular expression hack first, and... Lo and behold, it worked fine! The trick was to create two RE parsers, one to be used outside class declarations and one inside, and to switch them on the `class` keyword and counting braces.

Adding the methods to the relevant objects was actually simpler in C++ than in C, because the parser has all the information and does not need to depend on user-provided instructions.

Next we needed to handle name overloading, and we decided to cop out on that for the moment: duplicate names get "`_1`", "`_2`", etc appended for the time being. But we envision using the same trick as for constructors later.

Constructors turned out to map fairly well to the Python object model, except for multiple constructor signatures. This is the same problem as general name overloading, but here we really needed the overloading. It turns out that if is fairly easy to implement multimethods in a Python C extension module by doing a sequence of `PyArg_Parse` calls if you take some care in which order you do them:

```
if (PyArg_Parse(_args, "i", &ival)) {
      // body for method(int ival)
} else
if (PyArg_Parse(_args, "s", &sval)) {
      // body for method(const char *sval)
}
```

Support for `std::string` and such was just more code, nothing special. Other STL types we have punted on for the moment. Same for exceptions and operator overloading: they are not implemented yet but they should not pose any special problems (he says, optimistically:-).

At this point we had a fairly functional bridge that allowed calling C++ methods and functions from Python, and the next task was generating the reverse. We noted that all the relevant information was still in the generated Python file with the declaration info, so the only thing we needed to do was re-read it but putting the info in completely different implementations of `Module` and `ObjectDefinition`. These would then generate C++ to Python bridge code in stead of the reverse. And even this turned out not to be all that much work: passing an argument from Python to C++ is pretty similar to passing a return value from C++ to Python and vice versa, so most of the code was there, it only had be be refactored a bit and driven in a different way.

The same thing turned out to be true for generating the .h and .cpp files for the reverse bridge, which was again similar to generating the `PyTypeObject` and `PyMethodDef` structures and the actual implementations of the methods.

There were two areas that we did not have to worry about, luckily: constructors and non-function members. Our interfaces did not include constructors, all objects were constructed with factory functions [1]. Similar for non-function members, these are never part of interfaces. For the future, constructors are probably reasonably easy to implement, and for members we can probably use the Python slots mechanism in a similar way to what PyObjC does.

We did need one constructor per interface, so we could wrap Python objects in a C++ wrapper when passing it from Python to C++, but this was easy. The constructor got a Python object, did some rudimentary checking (the Python object needs to have attributes for all the needed C++ methods) and that was that.

Now the only thing needed was to tell the Python to C++ bridge about the available wrappers and we were all done: transparent passing of C++ objects to Python and the reverse. There were a few minor issues here, such as trying to make sure we did not do double wrapping (if a C++ object is passed to Python it gets a Python wrapper around it, but when we pass it back to C++ we want to pass the original object, not that Python wrapper wrapped in another C++ wrapper).

Results so far

This is all very recent work, so we have not had a chance to exercise it thoroughly, but we have managed to create a "non-player" in Python. This non-player goes through all the motions of reading, parsing and scheduling a document, but in stead of actually displaying media it prints messages of the form "at time hh:mm:ss I should have displayed image http://bla.bla.bla". Given the architecture of the Ambulant Player this means that all sorts of things, including subclassing both ways, threading, locking, and garbage collection, probably work.

We have also started to use the bridge to create a unit test suite for the Ambulant player. This is something we have sorely missed, but all C++ unit test frameworks appeared to be rather a lot of effort to deploy, so we are happy we can now use the Python unit test framework.

Future work

Some of the things that need to be done are sketched in the text, but one thing is conspicuously absent from this document, and really needs to be addressed: the weak points of bgen.

As you will have guessed by now bgen is not without them, being so mature but so little-used:-) The main problem is that the learning curve for deploying it in a new project, especially for the first time, is very steep. While its power resides in the ability to extend it any which way by subclassing the various `Type` and `Object` and `ObjectDefinition` classes this is also its downside: without that subclassing there is very little it can do. Similar for the RE parser: very powerful once you get the hang of it but usually the default parser will not work for your problem at hand without serious tweaking.

I have a number of ideas in this area, however. Actually, I have quite a large number of ideas, but I miss the accompanying amount of time. So let me end this paper with a cordial request to contact me if you feel like investing some of your spare time in *bgen*.

References

[1] http://pyobjc.sourceforge.net/

[2] http://www.ambulantplayer.org/

[3] http://www.swig.org/

[4] http://www.boost.org/libs/python/doc/

[1] In the Python to C++ bridge we did require the constructors, because the Python code may want to subclass specific C++ implementations of interfaces.

[5] http://www.riverbankcomputing.co.uk/sip/index.php

[6] http://www.python.org/peps/pep-0252.html

[7] http://www.python.org/peps/pep-0253.html

[8] http://www.gccxml.org/