Wrappy — a Python Wrapper Generator for C++ Classes

Gregory S. Couch
Conrad C. Huang
Thomas E. Ferrin

University of California
Computer Graphics Laboratory
San Francisco, CA 94143-0446
gregc@cgl.ucsf.edu

ABSTRACT

Wrappy generates a Python extension module from annotated C++ header files. Wrappy's power comes from how it translates C++ features to Python features: standard C++ container classes are mapped to Python lists and dictionaries; C++ classes are converted to Python types (or a Python-subclassable extension class [Ful98]); public member data and accessor member functions are mapped to attributes; C++ exceptions are mapped to Python exceptions; overloaded functions are supported; function argument names are used for Python keyword arguments. Annotating C++ header files is necessary to designate output arguments in functions and to choose how C++ classes are wrapped. Some C++ source code modifications are necessary to harmonize C++ and Python object semantics and to work around limitations in wrappy.

The impetus behind wrappy was to generate wrapper code for our (growing) C++ molecular graphics library with 500 member functions in 30 classes. Writing and maintaining wrapper code by hand for that many functions was unappealing. We also found that existing wrapper generators didn't support all the C++ features we use. Wrappy generates code that is similar to hand-crafted wrapper code, but more robust. The C++ header files generate about 30 times their ''weight'' in Python interface code (in our case, from 850 lines in C++ header files to over 25,000 lines of Python interface code).

1. Introduction

Wrappy is a programming tool that generates the wrapper code needed for a C++ library to be accessible from a Python interpreter. When combined, the wrapper code plus the C++ library become a Python extension module. In this paper we will discuss the reasons why we wrote this tool and how we mesh Python and C++ semantics.

1.1. A Bit of History

In the spring of 1997, we had a prototype of our next generation molecular modeling system that was written in C++ using Motif, OpenInventor, and OpenGL libraries. It was fast but cumbersome for prototyping work. Then we discovered Python and Tk.

By the spring of 1998, we had another prototype that was written in Python with Tk, OpenGL, and a thin custom C++ library (5 classes, 29 member functions, 4 global functions). It was easy to prototype new molecular modeling tools, but it was slower than the earlier prototype and at least 10 times slower than our previous generation of modeling software. We sped up the program by profiling the code and rewriting parts in C++ (especially I/O), but it was still too slow.

The challenge was to speed up the program dramatically while preserving the ease of prototyping that Python provides. Consequently, we increased the size of the the custom C++ library. The initial rewrite produced 30 C++ classes, ~500 member functions, and 3 global functions. Unlike the previous version of the C++ library, it

was not reasonable to write the Python wrapper code by hand. In the print of 1999, we wrote the **wrappy** program, an automated tool for wrapping C++ code with a Python interface in the spring of 1999.

2. Other Wrapper Generators

2.1. SWIG

Many people have worked on integrating C++ code with Python, which is a complex task as suggested by the continued presence of the Python Special Interest Group for C++[C++-SIG97]. Currently the most commonly used wrapping tool is SWIG 1.1[Bea97] from David Beazley at the University of Chicago. SWIG builds C/C++ extensions for several scripting languages. Although not designed specifically for C++ nor for Python, it works well with them.

SWIG supports many C/C++ features (simple class definitions, public constructors and destructors, virtual functions, public inheritance, global variables). However, SWIG does not support several C++ features that we use: function overloading, operator overloading, exceptions, namespaces, private destructors, standard C++ strings and containers.

Another problem with SWIG is that it generates inefficient code. SWIG uses a ‘‘shadow class’’ technique to export a C++ class into Python. For a C++ class, all member functions, including the constructors and destructors, are made into Python extension module functions. Those functions are encapsulated into a Python class by a companion Python script, thus shadowing the original C++ class. The advantage of shadow classes is that they are a simple means of allowing C++ classes to subclassed in Python. The disadvantages are that you need the companion script in addition to the extension module, and that class method invocation goes through a double dispatch before the C++ function is called. The double dispatch is costly as there is a significant overhead for the extra Python function call.

2.2. PYFFLE

Another wrapping tool that deserves special mention is PYFFLE[Mil98] from Patrick Miller at Lawrence Livermore National Laboratory. PYFFLE understands C++ better than SWIG. Its failings, for our purposes, were that it doesn’t parse C++ declarations to figure out what to wrap, that it is missing support for C++ exceptions and namespaces, and that it requires smart pointers for C++ resource management.

3. Making C++ Code Accessible from Python

3.1. Python and C++ Compatibility

One design goal was to get as much Python behavior out of our of C++ code as possible. Standard C++ and Python have many features in common. We were able to convert C++ exceptions to Python exceptions; convert standard C++ container classes to and from Python containers (by value); use C++ function argument names for Python function keyword arguments; convert C++ member data to Python class/type attributes; return C++ output function argument in Python tuples; *etc*. We also map C++ accessor member functions into Python type attributes, which makes the Python interface simpler than the C++ interface.

Where C++ and Python don’t match, the Python functionality is generally a subset of the C++ functionality. For instance, Python has no concept of a constant object. Here, wrappy depends on the compiler to give an error when it tries to compile the wrapper code.

3.1.1. Example: Accessor Functions as Attributes

An accessor function in C++ is a function that gets or sets the value of private member data. They are used to track changes in the member data and to keep a backwards-compatible interface during class design changes. For example, given the two C++ member functions:

```
std::string name() const;
void setName(
    const std::string &n);
```

Those two functions are combined into the Python attribute **name**. Getting the attribute’s value invokes the C++ *name* function, and setting the value invokes the C++ *setName* function.

3.2. Basic Ideas

As hinted above, the basic idea behind wrappy is that a C++ class should be made into a Python type. A Python type is different from a Python class: it is *not* subclassable. We can’t generate a Python class, because the Python class methods could not be C functions (no longer true in Python 1.5.2!). There is a third-party extension

to Python, Extension Class[Ful98] from Jim Fulton at Digital Creations, that provides the missing functionality. Wrappy optionally generates code that uses Extension Class if a C++ class needs to be subclassable in Python.

One advantage that the Python class mechanism has over the Python type mechanism is that the class methods are found by hashing and the type methods are found by linear search. For the Python primitive types, this has not been a problem because the number of methods is small, and they are placed in most commonly used order as determined by profiling. For the Python types that we generate from C++ classes, the number of methods is large (averaging 17 methods per C++ class), so we use a publicly available near-perfect hash generator, gperf[Sch98], to speed up method lookup.

Another basic idea is that wrappy should decide what Python interface to generate by parsing C++ declaration syntax. The C++ declarations are optionally annotated with special comments. For example, annotations are needed to designate output function arguments. These annotations don't work well for bulk removal of sections that should not be wrapped (*e.g.*, some member functions shouldn't be accessible to Python). The solution is to use only a subset of the the C++ header files, or embed ifdef's and use the C preprocessor to pipe the input to wrappy.

3.2.1. Example: Function Output Arguments

Sometimes a function returns information through some of its arguments. Often it is because it needs to return more than one value. Output arguments are annotated with an `OUT` comment. For example, the C++ function:

```
void bounds(const
    std::vector<int> &list,
    /*OUT*/ int *min,
    /*OUT*/ int *max);
```

is converted into a single argument Python function that returns a 2-tuple (min, max). The argument is checked to confirm that it is a sequence of integers and if not, an exception is raised.

4. Object Lifetimes (Reference Counting)

A difficult problem with wrapping C++ with Python is the dangling reference problem — insuring that C++ objects don't go away while there is a Python object that refers to it. Sometimes that is not possible, but we can prevent the dangling reference from crashing the Python interpreter.

Python avoids dangling references for its own objects by maintaining a reference count. When the reference count goes to zero, it is safe to remove the object. In C++, it is not so simple. Reference counting (via smart pointers) often is not used because the overhead is deemed to be unnecessary. Without reference counting, programmers depend on the documentation to determine how long objects are intended to stay around. Wrappy depends on annotations and code modifications to do the right thing.

4.1. Graph Example

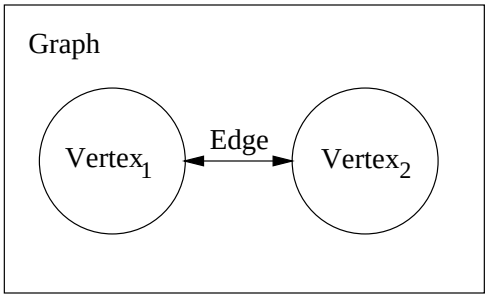

Figure 1

To illustrate some of the issues in wrapping C++ classes, consider a **Graph** class that manages **Vertex**es and **Edge**s (Figure 1). When a Graph goes away, all its Vertexes and Edges should be deleted as well. In C++, the way a class controls the lifetime of another class' objects is to allocate them on the heap. That restriction is enforced by the C++ compiler if the Vertex and Edge destructors are private, with the Graph class being a friend of Vertex and Edge, so it can call their destructors. Making these three classes available in Python requires careful management of Python references counts and C++ object lifetimes.

4.2. Class Wrapping Methods

Wrappy has two methods for associating a C++ object with a Python object. The first method is to put the C++ object inside a Python object by value. Here the C++ object and the Python object have identical lifetimes. When the Python object's reference count goes to zero, the C++ object is removed along with the Python object. This method works for any C++ class with a copy constructor, and works especially well for simple C++ objects that are passed by

value to other C++ functions.

The second method is to have the Python object keep a pointer to the C++ object. Here the C++ class must inherit from a particular base class known to wrappy. The base class keeps a pointer to the Python object that the C++ object belongs to. The Python object is lazily created, so a C++ object need not have a Python object associated with it. The works well for the Vertex and Edge classes above.

In the second method, a pointer from the C++ object back to the Python object is needed for two reasons. Every time the C++ object is converted to a Python object that it should return the same Python object. Secondly, if the C++ object goes away early, it can tell the Python object, so the Python object can throw an exception if it is used. Early removal of the C++ object should only happen when the C++ object's lifetime is controlled by the C++ code. Wrappy detects this case based on annotations and the presence of a private destructor. Wrappy gives the Python object an extra reference to prevent the reference count from going to zero. When the C++ object is deleted, it decrements the Python object's reference count which frees it. This method also works with private constructors.

4.3. Attribute Caching

There is still a synchronization problem between cross references among C++ objects and the reference counts of the corresponding Python objects. When a Python object's reference count goes to zero, it is removed and the memory reclaimed. The trick to synchronizing the reference count is to notice when the C++ object is keeping a reference to another C++ object. The technique is to cue off accessing attributes and other annotated functions. Consider the example:

```
define colorVertex(v):
    c = Color("green")
    v.color = c
```

Here a local object, *c*, is initialized with a new Python/C++ object and that object is assigned to an attribute of another hybrid Python object. Internally, the C++ Vertex object's color is set to the C++ Color object. If the reference count of the Python Color object isn't incremented by the assignment, when the local object goes away, its reference count will go to zero. Python will delete it and the associated C++ object, and the C++ Vertex object is left with a a dangling reference. A similar scenario exists for getting attributes. Consequently, whenever an attribute of object *P* is accessed, get or set to Python object *Q*, a reference to *Q* is saved in *P*. Saving the reference is known as attributes caching.

5. Future Work

We need to upgrade wrappy for Python 1.5.2 because Python 1.5.2 is a major improvement as a target for wrapper generators. It has extensive error checks for unimplemented number methods, so a Python type may implement only a subset of them. It also has a new N format for Py_BuildValue that returns a Python object without incrementing its reference count — a feature that we are already using. And the major change in Python 1.5.2, where Python class methods can be C functions, will enable us stop using the Extension Class extension. Being able to use Python classes for C++ classes means that we should revisit the design decision to use Python types (we should probably switch to using Python classes, but C++ classes that overload number methods or sequence methods might stay types).

There are other C++ features wrappy should support. It should be possible to support:

- alternative smart pointer mechanisms
- global variables
- template base classes
- explicitly instantiated template classes
- callback functions written in Python

These will be added as we need them.

6. Conclusion

Wrappy provides better support for standard C++ than SWIG and PYFFLE. Now, we can easily build new molecular modeling tools and get the speed we want. Python is used for the user interface, scripting, and writing modeling extensions. Anything speed critical can be rewritten in C++ and easily reintegrated into Python via a wrappy-generated extension.

Further information and source code can be found on the Internet at http://www.cgl.ucsf.edu/-Research/otf/wrappy/.

Acknowledgements

This work was funded by the NIH NCRR Resource for Biomolecular Graphics, NIH P41-RR01081.

This work would not have been possible without the help of our coworkers: Eric Pettersen,

Al Conde, Heidi Houtkooper, and Tom Goddard.

References

[Bea97] D. M. Beazley, *SWIG Users Manual, Version 1.1*, Department of Computer Science, University of Utah, Salt Lake City, Utah, June 1997. http://www.swig.org/.

[C++-SIG97] *C++-SIG -- SIG for Development of a C++ Binding to Python*, Corporation for National Research Initiatives, 1997-1999. http://www.python.org/sigs/.

[Ful98] J. Fulton, *Extension Classes, Python Extension Types Become Classes*, Digital Creations, Inc., 1998. http://www.digicool.com/-releases/ExtensionClass/.

[Mil98] P. Miller, *PYFFLE: Provides Your Fully Functional Logical Encapsulations*, Lawrence Livermore National Laboratory, 19 March 1998. http://www.python.org/pipermail/-c++-sig/1998-March/000071.html.

[Sch98] D. C. Schmidt, ‘‘Gperf: A Perfect Hash Function Generator,’’ *C++ Report*, 10(20):35-48, November/-December 1998.