Wrapping with SWIG and Boost.Python: a comparison

Prabhu Ramachandran

Department of Aerospace Engineering

IIT-Madras, Chennai, INDIA

Outline

- Introduction and how it works
- An example
- Wrapping
- Comparison
- Conclusions

Introduction

- Wrap C/C++ libraries to Python
- Require no modifications to existing code
- Support almost all C/C++ functionality
  - Basic data types, structs and classes
  - Pointers, references, smart pointers
  - Inheritance: override virtual functions in Python!
  - Function (and operator) overloading
  - Templates
  - Exceptions
  - Library Support (`std::vector`, `std::map` etc.)
- Provide a Pythonic interface

Introduction: SWIG

- 11 different target languages – Guile, Java, Mzscheme, OCAML, Perl, PHP, *Python*, Ruby, Tcl, Chicken and C#. More to come!
- Chief architect: David Beazley
- Around since around 1996
- Highly popular and default wrapping tool for a long while
- Version 1.1: incomplete C++ language support
- Version 1.3: *much* improved C++ support
- Implemented entirely in ANSI C
- Highly extensible

How it works: SWIG

- SWIG approach:
  - Parse C/C++ code
  - Generate low level wrapper code with ANSI C
  - Build wrapper to get low level module
  - Generate proxy classes (in target language) for the high level interface
- Director classes: target language virtual function overriding
- Typemaps: powerful, extensible, responsible for type conversions at language boundaries

Introduction: Boost.Python

- Only Python. More languages in future: langbind
- Chief architect: David Abrahams
- v1: 4th quarter 2000. v2: Oct. 2002.
- Part of very popular Boost C++ library
- Uses advanced C++ techniques
- Aim: Seamless Python/C++ integration
- Do the wrapping entirely from C++
- Clean C++ interfaces for wrapper generation
- Boilerplate code generation: *Pyste*

How it works: Boost.Python

- Boost.Python approach:
  - No parsing of C++ code
  - User writes simple wrapper code in pure C++
  - Template meta programs (boost::mpl) are used to do bulk of work underneath
  - Build wrappers to get Python module
- Provides high level C++ interface for manipulating Python objects
- Tries very hard to make the library usable safely from the interpreter
- Provides a seamless and Pythonic library

Introduction: Pyste

- Generates Boost.Python from Pyste files (`*.pyste`)
- Chief architect: Bruno da Silva de Oliveira (AKA: Nicodemus)
- First released: March 2003
- Pure Python code
- Pyste files are pure Python files
- New but getting better all the time
- Makes it easy to wrap large libraries
- Uses GCC_XML underneath to parse C++

How it works: Pyste

- Parse C++ using GCC_XML
- Generate Boost.Python code using parsed information
- Build wrappers to get Python interface
- Automates bulk of routine wrapping work
- Many useful features and is quite extensible
- Less powerful than doing everything in C++ directly

An example

- Non-trivial example
- Base class with pure virtual functions
- Subclass in C++ and Python
- Use `std::complex` and `std::vector`
- Consider function that accepts a `std::vector<T> &`
- Handle passing immutable objects by reference
- Test performance with different options

An example

```
// [ SNIP includes ]
class Vortex {
public:
    Vortex(const std::complex<double> &pos, const double str=1.0)
        : m_z(pos), m_s(str)
    { m_v = std::complex<double> (0.0, 0.0); }
    virtual ~Vortex() {}
// [ SNIP  setters and getters for m_z, m_s and m_v ]
    void addVelocity(const std::complex<double> &v)
    { m_v += v; }

    virtual std::complex<double>
    velocity(const std::complex<double> &pos) const = 0;

protected:
    std::complex<double> m_z, m_v;
    double m_s;
};
```

An example (contd.)

```
class PointVortex : public Vortex {
public:
    PointVortex(const std::complex<double> &pos,
                const double str=1.0) : Vortex(pos, str)
    {}

    virtual std::complex<double>
    velocity(const std::complex<double> &pos) const
    {
        if (pos != m_z)
            return -0.5*m_s/M_PI*(pos - m_z);
        else
            return std::complex<double>(0.0, 0.0);
    }
};
```

An example (contd.)

```
void computeVelocity(std::vector<Vortex*> &arr)
{
    for (std::size_t i=0; i<arr.size(); ++i)
        arr[i]->setVelocity(std::complex<double>(0.0, 0.0));
    Vortex *v1, *v2;
    for (std::size_t i=0; i<arr.size(); ++i) {
        v1 = arr[i];
        for (std::size_t j=0; j<arr.size(); ++j) {
            v2 = arr[j];
            v1->addVelocity(v2->velocity(v1->position()));
        }
    }
}

int multipleOutput(const int i, double &a, double &b)
{
    a += i;  b += i; return i;
}
```

Wrapping : SWIG

```
#ifdef DIRECTORS
%module(directors="1") example
%feature("director");
#else
%module example
#endif // DIRECTORS

%{ // header section
#include "example.hpp"
%}
%include "std_complex.i"
%include "std_vector.i"
%include "typemaps.i"
%apply double &INOUT { double &a, double &b };

%include "example.hpp"

%template(VectorVortex) std::vector<Vortex*>;
```

Wrapping : SWIG

- Building:

```
$ swig -c++ -python -DDIRECTORS -I. -o example_wrap.cxx example.i
$ c++ -Wall -O2 -I/usr/include/python2.2 -fPIC -I. -c \
  -o example_wrap.os example_wrap.cxx
$ c++ -shared -o _example.so example_wrap.os \
  -L/usr/lib/python2.2/config
```

- SWIG generates: `_example.so` and `example.py`
- `example.py` is the high level proxy interface
- Python: `import example`

Wrapping: Pyste

```
Vortex = Class("Vortex", "example.hpp")
PointVortex = Class("PointVortex", "example.hpp")
#final(PointVortex.velocity)
computeVelocity = Function("computeVelocity", "example.hpp")

mO_wrapper = Wrapper("mO_wrapper","""
object mO_wrapper(const int i, double a, double b)
{
    int ret = multipleOutput(i, a, b);
    return make_tuple(ret, a, b);
}""")
multipleOutput = Function("multipleOutput", "example.hpp")
set_wrapper(multipleOutput, mO_wrapper)

Include("boost/python/suite/indexing/vector_indexing_suite.hpp")
module_code("""
    class_<std::vector<Vortex*> > ("VectorVortex")
        .def(vector_indexing_suite<std::vector<Vortex*> > ());
""")
```

Wrapping: Boost.Python

```
#include <boost/python.hpp>
#include <boost/python/suite/indexing/vector_indexing_suite.hpp>
using namespace boost::python;

struct Vortex_Wrapper: Vortex {
    Vortex_Wrapper(PyObject* self_, const Vortex& p0):
        Vortex(p0), self(self_) {}

    Vortex_Wrapper(PyObject* self_, const complex<double>& p0):
        Vortex(p0), self(self_) {}

    Vortex_Wrapper(PyObject* self_, const complex<double>& p0,
                   const double p1): Vortex(p0, p1), self(self_) {}

    complex<double> velocity(const complex<double>& p0) const {
        return call_method< complex<double> >(self, "velocity", p0);
    }
    PyObject* self;
};
// SNIP PointVortex_Wrapper
```

Wrapping: Boost.Python (contd.)

```
BOOST_PYTHON_MODULE(example)
{
    class_< Vortex, boost::noncopyable, Vortex_Wrapper >("Vortex",
          init< const complex<double>&, optional< const double > >())
        .def("velocity", pure_virtual(&Vortex::velocity))
        .def("position", &Vortex::position)
        .def("myVelocity", &Vortex::myVelocity)
        .def("strength", &Vortex::strength)
        .def("setStrength", &Vortex::setStrength)
        .def("setPosition", &Vortex::setPosition)
        .def("setVelocity", &Vortex::setVelocity)
        .def("addVelocity", &Vortex::addVelocity)
    ;
    class_< PointVortex, bases< Vortex > , PointVortex_Wrapper >
          ("PointVortex", init< const PointVortex& >())
        .def(init< const complex<double>&, optional< const double > >())
        .def("velocity", &PointVortex::velocity,
            &PointVortex_Wrapper::default_velocity)
    ;
```

Wrapping: Boost.Python (contd.)

```
    def("computeVelocity", &computeVelocity);
    def("multipleOutput", &mO_wrapper);
    class_<std::vector<Vortex*> > ("VectorVortex")
        .def(vector_indexing_suite<std::vector<Vortex*> > ());
}
```

Wrapping: Boost.Python (contd.)

- Building:

```
$ pyste.py -I. --module=example example.pyste
$ c++ -Wall -O2 -ftemplate-depth-100 -DBOOST_PYTHON_DYNAMIC_LIB \
  -fno-inline -I/cvs/boost -I/usr/include/python2.2 -fPIC -I. \
  -c -o example.os example.cpp
$ c++ -shared -o example.so example.os \
  -L/cvs/boost/libs/python/build/bin-stage \
  -L/usr/lib/python2.2/config -lboost_python
```

- `example.so` is the Boost.Python module
- Recommended to build `example.cpp` with `bjam`

Comparison

Tool	Wrapper	Build	Linking	Total (secs)
SWIG	1.1	7.6	0.2	8.9
Pyste/BPL	21.0	46.0	0.8	67.8

Table 1: Build times

- Times on a PIII 450Mhz machine running Debian GNU/Linux (Woody)
- GCC version 2.95.4, Python 2.2
- SWIG module size: 88k; example.py: 8k
- Boost.Python module size: 1.3M

Comparison: performance

- Simple tests from Python
- Create several PointVortex objects (“Creation” column in Table 2)
- Compute velocity using computeVelocity (C++ function)
- Compute velocity using computeVelocity implemented in Python (Python)
- Subclass in Python and test (Subclass)
- Measure times

Comparison: performance

Option	Creation (secs)	C++ function	Python
SWIG (no-directors)	0.19	0.03	6.8
SWIG (directors)	0.19	0.03	7.3
SWIG (subclass)	0.19	5.47	9.3
BPL (no-wrappers)[*]	0.06	0.12	6.5
BPL (wrappers)	0.07	4.75	6.9
BPL (subclass)	0.07	6.82	9.1

Table 2: Performance for 300 vortices

[*] Don’t create Wrapper class; subclassing in Python will not work.

Conclusions

- Both tools successfully wrap the code and do a good job
- SWIG interface files and Pyste files: comparable sizes
- SWIG generates and build wrappers faster (10x)
- SWIG modules are smaller in size
- BPL does not require proxy classes
- Performance comparison: a few surprises but overall reasonable performance

Conclusions

- SWIG's lower level wrappers are accessible and it is possible to use the modules in an unsafe manner from the interpreter
- BPL provides very Pythonic and safe wrappers
- For large code bases (100 classes or more) SWIG should be faster to get up and running, slow compile time for BPL code can be a problem depending on compiler used
- The tools are all under constant development with an excellent developer community and great support
- All discussed tools do an excellent job and the future looks bright!