Simplified Wrapper Interface Generator

(aka SWIG)

Michel.Caillat@obspm.fr

Definition

- SWIG is a software development tool.
- It generates the code needed to bind C/C++ functions, classes, methods etc with scripting languages like Tcl, Python, Perl, Ruby, OCaml, MzScheme and also «non»-scripting languages like Java or C#.
- The C/C++ constructs to bind are described in an interface file which serves as input for SWIG to produce the wrapper.

History

- Created in 1995/1996 by David Beazley at LANL.
- Initial goal : «At the time, I was trying to add a data analysis and visualization capability to a molecular dynamics (MD) code...»
- At that time THE scripting language was Tcl/Tk , but quickly SWIG was designed to support Python, Perl and other.

A 1st Python example : the C++ side.

- Let's consider a 1st naive class C++ 'Square' defined as follows:

```
class Square {
private:
    double width;
public:
    Square(double w);
    virtual ~Square();
    double area() const;
    double perimeter() const;
};              // Square.h
```

- Assume that an implementation of Square exists in an object file Square.o

A 1st Python example: the SWIG interface.

- The class Square can be bound to Python via the following interface file :

```
%module Shapes
%{
#include "Square.h"
%}
class Square {
 private:
   double width;
 public:
   Square (double w);
   virtual ~Square();
   double area() const;
}; // Shapes.i
```

- Note that this binding ignores the method `perimeter`. Amongst other things the SWIG interface allows to select wich parts are to be bound.

- The name of the Python module to import is defined by '`%module Shapes`'.

A 1st Python example: the SWIG interface.

- The class Square can be bound to Python via the following interface file :

```
%module Shapes
%{
#include "Square.h"
%}
class Square {
 private:
   double width;
 public:
   Square (double w);
   virtual ~Square();
   double area() const;
}; // Shapes.i
```

Shapes = name of the python module to import

- Note that this binding ignores the method `perimeter`. Amongst other things the SWIG interface allows to select wich parts are to be bound.

- The name of the Python module to import is defined by '`%module Shapes`'.

A 1st Python example: the SWIG interface.

- The class Square can be bound to Python via the following interface file :

```
%module Shapes
%{
#include "Square.h"
%}
class Square {
 private:
   double width;
 public:
   Square (double w);
   virtual ~Square();
   double area() const;
}; // Shapes.i
```

Shapes = name of the python module to import

The method perimeter is not bound.

- Note that this binding ignores the method `perimeter`. Amongst other things the SWIG interface allows to select wich parts are to be bound.

- The name of the Python module to import is defined by '`%module Shapes`'.

A 1st Python example: building the wrapper.

- Before building the wrapper, assume we have in the current directory:

  `Square.h`, `Square.o`, `Shapes.i`

- Generate the code for the binding:

```
michel$ swig -c++ -python -o Shapes.cc Shapes.i
```

- Now we have in the current directory:

  `Square.h`, `Square.o`, `Shapes.i`, `Shapes.py` and `Shapes.cc`

- Build the wrapper:

```
michel$ g++ -I/usr/include/python2.6/ -fPIC \
-c Shapes.cc
michel$ gcc -shared -o _Shapes.so Shapes.o Square.o \
-lpython2.6 -lstdc++
```

- Now we have in the current directory:

  `Square.h`, `Square.o`, `Shapes.i`, `Shapes.py`, `Shapes.cc`, `Shapes.o` and `_Shapes.so` <--- the library loaded by Python

A 1st Python example: using the wrapper.

```
volte:~ michel$ ipython
Leopard libedit detected.
.
.
.

In [1]: import Shapes

In [2]: square = Shapes.Square(3.)

In [3]: square.area()
Out[3]: 9.0
```

- 'import Shapes' imports Shapes.py which itself :
  - imports the shareable library _Shapes.so which defines the C++ side of the binding.
  - defines the Python side of the binding.

SWIG Options (1)

- Numerous options to tailor the behaviour of swig :
  - `-c++` : specifies to process a c++ interface.
  - `-I<dir>` : where to look for the interfaces files.
  - `-o <outfile>` : defines the name of the file produced by swig.
  - `-outdir <dir>` : defines the directory where to write the file produced by swig.
  - `-help`
  - `-version`
- and many many others : ~ 70 in total.

SWIG Options (2)

- One option for each possible target language:

```
-allegrocl       - Generate ALLEGROCL wrappers
-chicken         - Generate CHICKEN wrappers
-clisp           - Generate CLISP wrappers
-cffi            - Generate CFFI wrappers
-csharp          - Generate C# wrappers
-guile           - Generate Guile wrappers
-java            - Generate Java wrappers
-lua             - Generate Lua wrappers
-modula3         - Generate Modula 3 wrappers
-mzscheme        - Generate Mzscheme wrappers
-ocaml           - Generate Ocaml wrappers
-octave          - Generate Octave wrappers
-perl            - Generate Perl wrappers
-php             - Generate PHP wrappers
-pike            - Generate Pike wrappers
-python          - Generate Python wrappers
-r               - Generate R (aka GNU S) wrappers
-ruby            - Generate Ruby wrappers
-sexp            - Generate Lisp S-Expressions wrappers
-tcl             - Generate Tcl wrappers
-uffi            - Generate Common Lisp / UFFI wrappers
-xml             - Generate XML wrappers.
```

swig -C++ -python. Inheritance (1)

- Let `Shape` be a «super class»...

```
class Shape {
 private:
  double xposition;
  double yposition;

 public:
  virtual double area() const = 0;
  virtual double perimeter() const = 0;
  void setPosition(double x, double y);
  double getXPosition() const;
  double getYPosition() const;
};
```

swig -C++ -python. Inheritance (2)

- ...of a class `Square`

```
class Square : public Shape {
 private:
  double width;

 public:
  Square(double width);
  virtual ~Square();
  virtual double area() const;
  virtual double perimeter() const;
};
```

swig -C++ -python. Inheritance (3)

- … and of a class `Circle`

```
class Circle : public Shape {
 private:
  double radius;

 public:
  Circle(double radius);
  virtual ~Circle();
  virtual double area() const;
  virtual double perimeter() const;
};
```

swig -C++ -python. Inheritance (4)

- Then with this SWIG interface :

```
%module Shape
%{
#include "Shape.h"
%}

class Shape {
 public:
  virtual double area() const = 0;
  virtual double perimeter() const = 0;
  void setPosition(double x, double y);
  double getXPosition() const;
  double getYPosition() const;
};

.../...
```

```
.../...

class Circle : public Shape {
 public:
  Circle(double width);
  virtual ~Circle();
  virtual double area() const;
  virtual double perimeter() const;

};
class Square : public Shape {
 public:
  Square(double width);
  virtual ~Square();
  virtual double area() const;
  virtual double perimeter() const;
}
```

Shape.i

swig -C++ -python. Inheritance (5)

- Then …

```
In [2]: from Shape import *

In [3]: issubclass(Circle, Shape)
Out[3]: True

In [4]: issubclass(Square, Shape)
Out[4]: True
```

swig -C++ -python. Inheritance (6)

- And also...

```
In [1]: from Shape import *
In [2]: c = Circle(1)
In [3]: c.area()
Out[3]: 3.1415926500000002
In [4]: c.perimeter()
Out[4]: 6.2831853000000004
In [5]: s = Square(3.)
In [6]: s.area()
Out[6]: 9.0
In [7]: s.perimeter()
Out[7]: 12.0
In [8]: s.setPosition(1., 2.)
In [9]: c.setPosition(3., 4.)
In [10]: print "s is at %f,%f" % (s.getXPosition(), s.getYPosition())
-------> print("s is at %f,%f" % (s.getXPosition(), s.getYPosition()))
s is at 1.000000,2.000000

In [11]: print "c is at %f,%f" % (c.getXPosition(), c.getYPosition())
-------> print("c is at %f,%f" % (c.getXPosition(), c.getYPosition()))
c is at 3.000000,4.000000
```

swig -C++ -python
Wrapping features (1)

SWIG has strategies to generate wrappers for these C++ constructs:

- ☑ functions,
- ☑ global variables,
- ☑ constants and enums,
- ☑ pointers,
- ☑ structures,
- ☑ classes and inheritance,
- ☑ overloaded functions, methods and ctors.

swig -C++ -python
Wrapping features (2)

And also :

- C++ operators (partly),
- C++ namespaces (be careful),
- C++ templates (actually instantiation of templates).

swig -C++ -python
template (1)

- Given this template definition

```
template<class T1, class T2> struct pair {
  T1 first;
  T2 second;
  pair();
  pair(const T1 &x, const T2 &y);
  template<class U1, class U2> pair(const pair<U1,U2> &x);
};

template<class T1, class T2>
pair<T1, T2>::pair():first(T1()), second(T2()) {}

template<class T1, class T2>
pair<T1, T2>::pair(const T1 &x, const T2 &y) : first(x), second(y)
{ }

template<class T1, class T2>
template<class U1, class U2>
pair<T1, T2>::pair(const pair<U1,U2> &x): first(x.first),second
(x.second) { }
```

swig -C++ -python
template (2)

- And this swig interface

```
%module pair
%{
#include "pair.h"
%}

template<class T1, class T2> struct pair {
  T1 first;
  T2 second;
  pair();
  pair(const T1 &x, const T2 &y);
  template<class U1, class U2> pair(const pair<U1,U2> &x);
  %template(pairc) pair<int, int>;
};


%template(pairii) pair<int, int>;
```

pair.i

swig -C++ -python
template (2)

- And this swig interface

```
%module pair
%{
#include "pair.h"
%}

template<class T1, class T2> struct pair {
  T1 first;
  T2 second;
  pair();
  pair(const T1 &x, const T2 &y);
  template<class U1, class U2> pair(const pair<U1,U2> &x);
  %template(pairc) pair<int, int>;
};


%template(pairii) pair<int, int>;
```

Only the instantiation `pair<int, int>` of pair will be callable from Python under the name `pairii`

pair.i

swig -C++ -python
template (2)

- And this swig interface

```
%module pair
%{
#include "pair.h"
%}

template<class T1, class T2> struct pair {
  T1 first;
  T2 second;
  pair();
  pair(const T1 &x, const T2 &y);
  template<class U1, class U2> pair(const pair<U1,U2> &x);
  %template(pairc) pair<int, int>;
};


%template(pairii) pair<int, int>;
```

pair.i

The (template) copy constructor will have the name `pairiic` !

Only the instantiation `pair<int, int>` of pair will be callable from Python under the name `pairii`

swig -C++ -python
template (3)

- and play in Python :

```
In [1]: import pair

In [2]: p = pair.pairii(3, 4)

In [3]: print "p = %d,%d" % (p.first, p.second)
------> print("p = %d,%d" % (p.first, p.second))
p = 3,4

In [4]: q = pair.pairiic(p)

In [5]: print "q = %d, %d" % (q.first, q.second)
------> print("q = %d, %d" % (q.first, q.second))
q = 3, 4

In [6]: q.first = -8

In [7]: print "q = %d, %d" % (q.first, q.second)
------> print("q = %d, %d" % (q.first, q.second))
q = -8, 4

In [8]: print "p = %d,%d" % (p.first, p.second)
------> print("p = %d,%d" % (p.first, p.second))
p = 3,4
```

swig -C++ -python %extend (1)

- Use %extend to add new methods «on the fly» in the interface.

```
%module Shape
%{
#include "Shape.h"
%}

class Shape {
 public:
  virtual double area() const = 0;
  virtual double perimeter() const = 0;
  void setPosition(double x, double y);
  /* double getXPosition() const;
   double getYPosition() const; */
};
%extend Shape {
   double xPosition() const {
   return $self->getXPosition();
   }
   double yPosition() const {
   return $self->getYPosition();
   }
};
```

.../...

```
.../...

class Circle : public Shape {
 public:
  Circle(double width);
  virtual ~Circle();
  virtual double area() const;
  virtual double perimeter() const;

};
class Square : public Shape {
 public:
  Square(double width);
  virtual ~Square();
  virtual double area() const;
  virtual double perimeter() const;
}
```

Shape.i revisited

swig -C++ -python %extend (1)

We replace the «natural» methods get(X|Y)Position by ....

- Use %extend to add new methods «on the fly» in the interface.

```
%module Shape
%{
#include "Shape.h"
%}

class Shape {
 public:
  virtual double area() const = 0;
  virtual double perimeter() const = 0;
  void setPosition(double x, double y);
  /* double getXPosition() const;
   double getYPosition() const; */
};
%extend Shape {
   double xPosition() const {
   return $self->getXPosition();
   }
   double yPosition() const {
   return $self->getYPosition();
   }
};
```

.../...

```
.../...

class Circle : public Shape {
 public:
  Circle(double width);
  virtual ~Circle();
  virtual double area() const;
  virtual double perimeter() const;

};
class Square : public Shape {
 public:
  Square(double width);
  virtual ~Square();
  virtual double area() const;
  virtual double perimeter() const;
}
```

Shape.i revisited

swig -C++ -python %extend (1)

We replace the «natural» methods get(X|Y)Position by ....

...two methods (x|y)Position

- Use %extend to add new methods «on the fly» in the interface.

```
%module Shape
%{
#include "Shape.h"
%}

class Shape {
 public:
  virtual double area() const = 0;
  virtual double perimeter() const = 0;
  void setPosition(double x, double y);
  /* double getXPosition() const;
   double getYPosition() const; */
};
%extend Shape {
   double xPosition() const {
   return $self->getXPosition();
   }
   double yPosition() const {
   return $self->getYPosition();
   }
};
```

.../...

```
.../...

class Circle : public Shape {
 public:
  Circle(double width);
  virtual ~Circle();
  virtual double area() const;
  virtual double perimeter() const;

};
class Square : public Shape {
 public:
  Square(double width);
  virtual ~Square();
  virtual double area() const;
  virtual double perimeter() const;
}
```

Shape.i revisited

swig -C++ -python %extend (1)

We replace the «natural» methods get(X|Y)Position by ....

...two methods (x|y)Position

- Use %extend to add new methods «on the fly» in the interface.

```
%module Shape
%{
#include "Shape.h"
%}

class Shape {
 public:
  virtual double area() const = 0;
  virtual double perimeter() const = 0;
  void setPosition(double x, double y);
  /* double getXPosition() const;
   double getYPosition() const; */
};
%extend Shape {
   double xPosition() const {
   return $self->getXPosition();
   }
   double yPosition() const {
   return $self->getYPosition();
   }
};
```

$self is the keword to refer to the current instance.

.../...

```
.../...

class Circle : public Shape {
 public:
  Circle(double width);
  virtual ~Circle();
  virtual double area() const;
  virtual double perimeter() const;

};
class Square : public Shape {
 public:
  Square(double width);
  virtual ~Square();
  virtual double area() const;
  virtual double perimeter() const;
}
```

Shape.i revisited

swig -C++ -python %extend (2)

- And in python :

```
In [2]: import Shape

In [3]: s = Shape.Square(6)

In [4]: s.setPosition(-8, -8)

In [5]: print "s is at %f,%f" % (s.xPosition(), s.yPosition())
------> print("s is at %f,%f" % (s.xPosition(), s.yPosition
())))
s is at -8.000000,-8.000000

In [6]:
```

swig -C++ -python %extend (2)

- And in python :

```
In [2]: import Shape

In [3]: s = Shape.Square(6)

In [4]: s.setPosition(-8, -8)

In [5]: print "s is at %f,%f" % (s.xPosition(), s.yPosition())
------> print("s is at %f,%f" % (s.xPosition(), s.yPosition
())))
s is at -8.000000,-8.000000

In [6]:
```

Now we call the methods `xPosition` and `yPosition` defined in the interface.

swig -C++ -python %extend (3)

- Useful to customize the behaviour of the standard «special» Python methods on the C ++ bound objects like __str__(), __repr__(), __hash__(),__cmp__(), __eq__(), and other comparisons methods.

swig -C++ -python %extend (4)

- Example: customization of __str__() :

```
%extend Circle {
  char* __str__() {
    static char temp[100];
    sprintf(temp,
"I am a circle with a radius of %f and I am located at %f,%f",
$self->radius(), $self->getXPosition(), $self->getYPosition());
    return &temp[0];
  }
}
```

swig -C++ -python %extend (4)

- Example: customization of __str__() :

Inserted in Shape.i after the definition of the interface of Circle

```
%extend Circle {
  char* __str__() {
    static char temp[100];
    sprintf(temp,
"I am a circle with a radius of %f and I am located at %f,%f",
$self->radius(), $self->getXPosition(), $self->getYPosition());
    return &temp[0];
  }
}
```

swig -C++ -python %extend (4)

- Example: customization of __str__() :

Inserted in Shape.i after the definition of the interface of Circle

```
%extend Circle {
  char* __str__() {
    static char temp[100];
    sprintf(temp,
"I am a circle with a radius of %f and I am located at %f,%f",
$self->radius(), $self->getXPosition(), $self->getYPosition());
    return &temp[0];
  }
}
```

```
In [1]: import Shape
In [2]: c = Shape.Circle(5)
In [3]: c.setPosition(-2, -2)
In [4]: print c
------> print(c)
I am a circle with a radius of 5.000000 and I am located at
-2.000000,-2.000000
```

swig -C++ -python %include and %import

- %include : includes predefined interfaces in your interface file.
  (remember the option -I ?).

```
%include “foo.i“
%include “std_vector.i“
```

- %import : just let the interface to know about another interface without generating code for it. (useful for modules having cross references).

SWIG library (1)

- SWIG comes with a library of predefined interfaces to facilitate the binding of some standard C/C++ constructs.
- Use the `%include` statement to incorporate these interfaces into your interfaces.

```
volte:4thexample michel$ swig -swiglib
/opt/local/share/swig/1.3.40
```

SWIG library (1)

- SWIG comes with a library of predefined interfaces to facilitate the binding of some standard C/C++ constructs.
- Use the `%include` statement to incorporate these interfaces into your interfaces.

```
volte:4thexample michel$ swig -swiglib
/opt/local/share/swig/1.3.40
```

Where is the SWIG library ?

SWIG library (2)

- C SWIG library :
  - `%include “cpointer.i“`
  - `%include “carrays.i“`
  - and others…
- C++ SWIG library:
  - `%include “std_string.i“`
  - `%include “std_vector.i“`
  - `%include “std_map.i“`
  - `%include “std_vector.i“`
  - `%include “std_set.i“`
  - `%include “std_list.i“`

swig -C++ -python
example of use of std_vector.i (1)

- The C++ definition of a class MyVec:

```
#include <vector>

class MyVec {
 public:
  static double module (const std::vector<double> & v);
  static std::vector<double> drand(int n, double x0, double x1);
};
```

swig -C++ -python
example of use of std_vector.i (2)

- Its SWIG interface

```
%module MyVec
%include "std_vector.i"
%{
#include "MyVec.h";
  %}

%template(vdble) std::vector<double>;

class MyVec {
 public :
  static double module (const std::vector<double>& v) const;
  static std::vector<double> drand(int n, double x0, double x1);
};
```

swig -C++ -python
example of use of std_vector.i (2)

- Its SWIG interface

include the SWIG interface of std::vector

```
%module MyVec
%include "std_vector.i"
%{
#include "MyVec.h";
  %}

%template(vdble) std::vector<double>;

class MyVec {
 public :
  static double module (const std::vector<double>& v) const;
  static std::vector<double> drand(int n, double x0, double x1);
};
```

swig -C++ -python
example of use of std_vector.i (2)

- Its SWIG interface

include the SWIG interface of std::vector

Don't forget to rename std::vector<double>

```
%module MyVec
%include "std_vector.i"
%{
#include "MyVec.h";
  %}

%template(vdble) std::vector<double>;

class MyVec {
 public :
  static double module (const std::vector<double>& v) const;
  static std::vector<double> drand(int n, double x0, double x1);
};
```

swig -C++ -python
example of use of std_vector.i (3)

- Use it in Python :

```
In [1]: import math

In [2]: import MyVec

In [3]: print MyVec.MyVec_module([math.sin(0.5), math.cos(0.5)])
------> print(MyVec.MyVec_module([math.sin(0.5), math.cos(0.5)]))
1.0

In [4]: print MyVec.MyVec_drand(5, 0., 1.)
------> print(MyVec.MyVec_drand(5, 0., 1.))
(0.83855384906686559, 0.57454126680946971, 0.31507126675689184,
0.40278038308153888, 0.52989845142229386)

In [5]:
```

swig -C++ -python
example of use of std_vector.i (3)

- Use it in Python :

```
In [1]: import math

In [2]: import MyVec

In [3]: print MyVec.MyVec_module([math.sin(0.5), math.cos(0.5)])
------> print(MyVec.MyVec_module([math.sin(0.5), math.cos(0.5)]))
1.0

In [4]: print MyVec.MyVec_drand(5, 0., 1.)
------> print(MyVec.MyVec_drand(5, 0., 1.))
(0.83855384906686559, 0.57454126680946971, 0.31507126675689184,
0.40278038308153888, 0.52989845142229386)

In [5]:
```

Note the prefix `MyVec_` prepended to each method name. This is because they are static.

Exceptions

- SWIG is able to bind exceptions, i.e. a declaration :

```
someMethod(...) throw(SomeExceptionClass);
```

in the SWIG interface , allows a

```
try :
  invoke someMethod(...)
except SomeExceptionClass, e:
  # e is a wrapped instance of SomeExceptionClass
```

in the Python code.

Personal feeling.

- SWIG is easy to use to perform simple bindings.
- SWIG is richly documented. This is precious when more complex bindings must be done.
- SWIG should be considered for a project involving bindings of C/C++ libs.
- SWIG has satisfied all my needs in terms of binding.
- Questions :
  - Quality of the code generated ?
  - Potential dangers (e.g. memory leaks) ?