Combining C++ and python using SWIG

Asa Ben-Hur

What is SWIG?

- A tool for interfacing various programming languages to C/C++ (supported languages: C#, chicken, lisp, guile, java, lua, scheme, Ocaml, perl, php, python, ruby, Tcl/Tk)
- Requires no modifications to existing code (in most cases).

Why use SWIG?

- Some tasks are easier in scripting languages:
  - User interface
  - Testing
  - Customize and reconfigure with recompilation
- Incorporating C/C++ into a higher level language often results in a more modular design, less code, better flexibility, and increased programmer productivity
- C programming becomes more enjoyable (or tolerable).

Ways of using SWIG

- Building more flexible C/C++ programs.
  - Replace main() with a scripting interface from which you can control the application. Modify the behavior of the program without having to modify low-level C/C++ code.
  - Rapid testing and debugging.
    - Test a library with a collection of scripts
    - Use the interpreter as an interactive debugger.

Ways of using SWIG (cont)

- Systems integration.
  - Scripting languages work well for gluing loosely-coupled software components.
  - Building high performance modules for scripting languages.

Language features supported by SWIG

- Almost any C/C++ construct can be wrapped (exceptions: nested class definitions).
- Support for STL.
- Some features cannot be encapsulated by the scripting language (e.g. multiple inheritance)
A simple SWIG interface file

```c
%module mymodule
%
#include "myheader.h"
%
// Now some C/C++ declarations:
int myvariable;
int myfunction(int x);
...
```

Naming the extension module

The `{…}` block is copied to the resulting wrapper file.

// Now some C/C++ declarations:
int myvariable;
int myfunction(int x);
...

Building a Python module

```bash
swig -python example.i
gcc -c -fpic example.c example_wrap.c 
    -I/usr/local/include/python2.5
gcc -shared example.o example_wrap.o -o _example.so
``` 

In python:

```python
>>> import example
>>> example.fact(4)
24
``` 

Pointers

- Pointers are fully supported by SWIG
- SWIG representation of a pointer:
  `<Swig Object of type 'double *' at 0x430069>`
- All pointers are treated as opaque objects by SWIG. Thus, a pointer may be returned and passed around to other C functions as needed. For all practical purposes, the scripting language interface works in exactly the same way as you would use the pointer in a C program.
- There is no mechanism for dereferencing the pointer since this would require the target language to understand the memory layout of the underlying object.

Supported data types

- Primitive types such as `int` and `double` are mapped to corresponding types in the target language.
- Everything else is a pointer

Everything is a pointer

- When SWIG encounters an undeclared type, it assumes that it is a structure or class. E.g.: `void matrix_multiply(Matrix *a, Matrix *b, Matrix *c);`
- SWIG has no idea what a "Matrix" is. However, it is obviously a pointer to something so SWIG generates a wrapper using its generic pointer handling code. This allows SWIG to generate interfaces from partial information. In some cases, you may not care what a Matrix really is as long as you can pass a reference to one around in the scripting language interface.

Supported structures and classes

A class such as:

```c
class Vector {
    Vector();
    ~Vector();
    double x, y, z; };
``` 

can be wrapped using the following set of functions:

```c
Vector *new_Vector();
void delete_Vector(Vector *v);
double Vector_x_get(Vector *v);
void Vector_x_set(Vector *v, double x);...
```
Using the Class

```python
>>> v = new_Vector()
>>> Vector_x_set(v, 3)
>>> Vector_delete(v)
```

Using Proxy/Shadow Classes

A more natural way of wrapping:

```python
>>> v = Vector()
>>> v.x = 3
>>> ...
>>> del v
```

The shadow class is a python class that wraps the accessor functions

```
swig -python -shadow -c++ example.i
```

The Vector python shadow class

```python
class Vector:
    def __init__(self):
        self.this = new_Vector()  # this is a pointer to the c++ object
    def __del__(self):
        delete_Vector(self.this)
    def length(self, x):
        return Vector_length(self.this)
    def __getattr__(self, name):
        if name == 'x':
            return Vector_x_get(self.this)
        ...
    def __setattr__(self, name, value):
        if name == 'x':
            Vector_x_set(self.this, value)
        ...
```

Function overloading

```python
class Foo:
    public:
        Foo();
        Foo(const Foo &);   // Copy constructor
    
>>> f = Foo()      # Create a Foo
>>> g = Foo(f)      # Copy Foo
```

- SWIG determines the proper function at runtime
- Ambiguities can occur!

Operator overloading

- Supported as well Example: a Complex class:
  ```python
  >>> a = Complex(3,4)
  >>> b = Complex(5,2)
  >>> c = a + b
  ```
- Caveat: the equality operator
- Used to be a little more complicated requiring:
  ```
  $rename{__add__} Complex::operator+;
  ```

Memory management

```python
class Foo:
    public:
        Foo();
        Foo(const Foo &);
        Foo bar();
        
>>> f = Foo()        # Create a Foo
>>> g = f.bar()       # Copy Foo
```

The value of the ownership flag (.thisown) determines who is responsible for deleting the underlying c++ object
Memory management

class Foo {
public:
  Foo();
  Foo *spam();
-};

>>> f = Foo()
>>> g = f.spam()
>>> g.thisown
False

Templates

- Are supported: need to wrap a template instantiation (e.g., vector<int>).
- An instantiation name such as vector<int> is generally not a valid identifier name in most target languages. Give the instantiation a more suitable name such as intvector.

The SWIG library

- Contains support for:
  - C arrays and pointers
  - C strings
  - STL

%module example
-
%include "carrays.i"
%array_class(int, intArray);
%array_class(double, doubleArray);

Which can be used in python as:

def doubleArray(a):
  array = example.doubleArray(len(a))
  for i in range(len(a)):
    array[i] = a[i]
  return array

C arrays

%include "std_string.i"
%iinclude "std_vector.i"
namespace std
{
%template(IntVector) std::vector<int>;
%template(DoubleVector) std::vector<double>;
%template(StringVector) std::vector<string>;
}

STL

The declaration:

#include "std_string.i"
#include "std_vector.i"

Provides infrastructure for converting C++ std::string objects to and from strings in the scripting language

- For strings only std::string and const std::string & are supported.

STL

%include "std_string.i"
%iinclude "std_vector.i"
Can also do:

%include "stl.i"
namespace std
{
%template(IntVector) std::vector<int>;
%template(DoubleVector) std::vector<double>;
%template(StringVector) std::vector<string>;
}

- A class that exposes the C++ API is created in the target language
- Input typemaps are defined for vector<X>, const vector<X> & , and const vector<X> *
- An output typemap is defined for vector<X>
Interface files

- SWIG doesn't care if the declaration of a structure in a .i file exactly matches that used in the underlying C/C++ code. You can omit problematic members or omit the structure or class definition altogether (if you're happy passing pointers around).

What to do with main()

- Whenever writing a library that is intended to be wrapped, main() should only contain argument parsing code.
- Put main() in a separate file. In fact, if the library is to be used via a scripting language interface - you don't even need one.

Typemaps

```cpp
void add(double a, double b, double *result) {
  *result = a + b;
}
```

- Reading the code, it is clear that *result is the output of the function. How can we make SWIG aware of this?

```plaintext
#include "typemaps.i"
%apply double OUTPUT { double *result }

>>> c = add(a, b)
```

Advanced topics

- C/C++ helper functions (%inline)
- Inserting python code (%pythoncode)
- Extending a C++ class (%extend)
- Exception handling (%exception)
- Docstrings
- Smart pointers
- Reference counting

For future reference

- The SWIG website: [http://www.swig.org](http://www.swig.org)
- The lecture material: [http://www.cs.colostate.edu/~asa/swig](http://www.cs.colostate.edu/~asa/swig)