

Using Python with Other Languages

Vishesh Yadav | Siddhant Sanyam

PyCon India 2011

09 September 2011

1 Python With C++

2 Extending C++ using Boost.Python

3 Extending Python with C and C++ using SWIG

Why C++ ?

- Because it is as good as C (well, most of the time)
- Extending C API is ugly, and terribly confusing
- Because it is Object Oriented
 - Thus you can think in OOP
 - Resembles Python (the OO part)
- It is more powerfull as a language
- Offers competitive performance as C
- Because of Boost (wait till I introduce you to it)

What can be done with Python and C++ ?

- Same stuff you can do with C but more
- You can have class abstractions
- Extending Python with C++
- Embedding Python in C++
- Python is more close to C++ than to C

Available options to extend Python with C++

There are multiple options that are available to extend Python with C++

- Using the ever-friendly Python C API
- Boost's API for C++
- SWIG Let's take each one of them one by one

Extending Python with C++ Using Python C API

- Basic idea is that since C++ is essential “C with Classes”, we can use C++ as C and use the Python C API
- But that sucks! ... at multiple levels
 - You have to deal with the ugly API and you miss the comfort of Boost and SWIG
 - You are using C++ as C, so no classes and objects, why not use C then?
 - For simple cases when you want to enjoy C++ as a language (with STL and all), it makes sense
 - But you can't expose the class-like interface of C++
- When will you use it? Almost never. Why? Because there are superior alternative available

Introduction to Boost.Python

So basically, it is a wrapper class on the Python C API It's awesome

- Because it is easy to use (majorly because of the syntax)
- You can expose the class interface of C++ with this
- Great for small to medium sized projects
- It does operator overloading too!
- Inheritance is also supported
- In short, it brings C++ almost completely to Python

Boost.Python kickstart

We assume that you have Boost.Python installed on your system.

- Boost.Python also works without installing it, although it is recommended that you do install it.
- Boost.Python requires you to include a simple header file `<boost/python.hpp>`
- We'll be basically writing Python modules using C++

Compiling and Build process

You have many options to build and compile Boost

- bjam, the recommended way of building Boost
- manually linking and compiling using g++

We'll assume g++ to be de facto standard compiler. Python 2.4 and above is recommended.

Hello Boost!

Our purpose is to write a module `hello` which has a function called `say`

- We'll writing the definition of the function `say` in C++
- Then we'll use Boost.Python to expose the function in a module called `hello`
- It's similar stuff we did with Python C API, but I think you'll appreciate the neatness.

Here is the code for say

say takes one unsigned int as an argument and returns 'Hello', 'Boost.Python' or 'World' depending on if the int passed was 0, 1 or 2. Definition of say

```
1: char const* say(unsigned x)
2: {
3:     static char const* const msgs[] = { "Hello", "Boost.Py
4:
5:     if (x > 2)
6:         throw std::range_error("say: index out of range");
7:
8:     return msgs[x];
9: }
```

- Simple enough right?
- Well, writing C++ is always the simple part, the important part is making a module out of it.

The Boost's Magic

Turns out that to write the module `hello` and putting function `say` in it is pretty easy in `Boost.Python`

```
1:  #include <boost/python.hpp>
2:  using namespace boost::python;
3:  BOOST_PYTHON_MODULE(hello)
4:  {
5:      def("say", //the name of the function to be exposed in
6:          say,   //the function name in c++
7:          "return one of 3 part of Hello Boost.Python World"
8:  }
```

That's it!

- Now you know what is meant by “easy to use”

Compile and Go

As I said there are multiple way to compile and go. Easiest is through bjam Just execute the command

```
bjam --tools-set
```

Now that's cool, but what about higher stuff?

- We'll take each example file one by one
- Please use a text editor with Line numbering because we'll be using the line numbers to reference examples

Examples

- Exposing Classes
- Operator Overloading
- Inheritance
- Virtual Function
- Serialization
- Object coversion

SWIG

It's a wrapper and interface writer for C/C++ It means

- You write C or C++ code, and this tool will write the interface and wrapper to other languages for your code.
- It supports many languages like Python, Tcl, Ruby, Guile, and Java
- It supports nearly every feature of C++
- Usually it offers a neat distinction between the original code and the interface (thus you don't have to modify your original sources much)

Advantage of SWIG

- It's a uniform over different language
- Hence it is awesome way to have wrappers for your C++ library.
- It's much more complete to cover features of C++
- If your interface is in separate file, writing SWIG interface is trivial.
- It requires none or very less change in your original source code.

How does it work for Python

- You first write your C/C++ interface and definition
- You write a dot i file (example.i) which specifies the interface of the wrapper
- Use SWIG to generate wrapper files of Python
- Compiler C/C++ code to a shared library
- Use your freshly baked module.

SWIG kickstart

- You should be having SWIG installed, Python installed.
- You optionally would like to have a c++ compiler installed :P
- We'll be using our beloved GNU's GCC in examples.

Hello SWIG

First create the C part Header file/Interface file

```
1:  /*hello.h*/  
2:  
3:  /*Signature, Prototype, Definition of the function*/  
4:  int factorial(int n);
```

The definition

Now we'll write the definition of our function

```
1:  /* hello.c */
2:  /* Definition of factorial */
3:  #include "hello.h"
4:  int factorial(int n) {
5:      if (n < 0){
6:          return 0; // Hmm, we're treating this error case
7:      }
8:      if (n == 0) {
9:          return 1;
10:     }
11:     else {
12:         return n * factorial(n-1);
13:     }
14: }
```

Using in a C program

This is how the main file will use this library in your typical C program

```
1:  #include <stdio.h>
2:  #include "hello.h"
3:
4:  int main(void)
5:  {
6:      int a = 5;
7:      printf("%i", factorial(a) );
8:  }
```

And then you compile it with hello.c

```
gcc main.c hello.c -o hello_executable
./hello_executable
```

This was C, let's make it Pythonable

- You will appreciate the simplicity of SWIG.
- Our C interface is already done.
- We'll just write a SWIG's .i file to expose the interface to SWIG

hello.i

```
/* File: hello.i */
%module hello

/*
    The following blocks are included as is in the wrapper file.
    Important to include the interface file in wrapper file
    because the input file is just used for generating the wrapper.
*/

%{
#include "hello.h"
%}

/*
    We'll now declare the interface for the wrapper file.
    At worst case, we can simply #include "hello.h" again
*/
int factorial(int):
```


Let it SWIG

Now will run SWIG on this input file we just wrote.

```
swig -python hello.i
```

Now you should have these files

```
hello.py hello.h hello.c hello_wrap.c
```

Compile your stuff

```
gcc -c hello.c hello_wrap.c -I /usr/include/python2.7 -lpython2.7
```

You'll be having `hello.o` and `hello_wrap.o` . It's time to link them into a shared library The name of the so file is module name prefixed by an underscore

```
ld -shared -o _hello.so hello.o hello_wrap.o
```

Ta da! we have got our `_hello.so` ready.

Lock and load

- Now we have `hello.py` (generated by SWIG using `hello.i`)
- We can simply `import hello` to load the module.
- The loading of the `.so` file is job of `hello.py`

Using Module

```
python  
>>> import hello  
>>> hello.factorial(5)  
120  
>>>
```

- Enjoyed it! Let's do it for C++ now

First thing first, writing the C++ code

- We could all the above example of simple function exposing in C++ too
- But since it is C++, We'll do OOP by having a simple class of Point number

The interface file

```
/* File : Point.h */  
  
class Point  
{  
private:  
    int x;  
    int y;  
public:  
    Point(int,int);  
    float dist_origin(); //distance from origin  
};
```

Implementation file for Point

```
/* File: Point.cpp */
#include "Point.h"
#include <cmath>
Point::Point(int X, int Y)
{
    x=X;
    y=Y;
}
float Point::dist_origin()
{
    return pow(x*x + y*y, 0.5);
}
```

Using this Library in C++ (normally)

```
/* File: main.cpp */  
#include <iostream>  
#include "Point.h"  
int main()  
{  
    Point p1= Point(2,3);  
    std::cout << p1.dist_origin();  
}
```

Let's compile this, like we do it in C++

```
g++ main.cpp Point.cpp -o point_example  
./point_example
```

Now time for it to be eaten by Python

We'll write the SWIG's .i file

```
/* File : Point.i */
```

```
%module Point // Module name
```

```
%{
```

```
#include "Point.h" //including it for the Point_wrap.cxx file
```

```
%}
```

```
/*
```

```
We are being lazy over here. Instead of putting the  
actual interface, we just copied it from Point.h
```

```
*/
```

```
%include "Point.h"
```

Let's quickly make the wrapper c++ file and the interface .py files

Compiling the resultant files

The following file are generated

```
Point_wrap.c Point_wrap.cxx Point.py
```

Let's use g++ to compile our wrapper and source file

```
g++ -O2 -fPIC -c Point.cpp Point_wrap.cxx -I /usr/include
```

Now we'll make the shared library `_Point.so` (note the naming convention is same)

```
g++ -shared -o _Point.so Point.o Point_wrap.o
```

That's it, we can start using this class in our Python code

Run the module

```
python  
>>> import Point  
>>> p = Point.Point(3,4)  
>>> p.dist_origin()  
5.0
```

Wasn't that simple?

SWIG Examples

- Global Variables
- SWIG Directives
- Constants
- Pointers
- Operator Overloading
- Inheritance