# An introduction to the Simplified Wrapper and Interface Generator (SWIG)

Prabhu Ramachandran

Department of Aerospace Engineering
IIT Bombay

18, July 2007





## Outline

1. Introduction
2. Usage
3. Interface files

## Introduction

- High performance codes often written as C/C++ libraries
- This slows down development, prototyping, and experimentation
- Much easier to use (and glue) if Python interface exists
- SWIG lets you wrap C/C++ libraries to various target languages
- SWIG is Open Source (BSD-like license)





## Introduction . . .

- 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
- Extensible and configurable through the use of typemaps

## Features

- Basic data types, structs and classes
- Pointers, references, smart pointers
- Functions
- Inheritance: override virtual functions in Python!
- Function (and operator) overloading
- Templates
- Exceptions
- Library Support (`std::vector`, `std::map` etc.)
- NumPy support through `numpy.i`





## Resources

- Available for most GNU/Linux distributions
- MacPorts for the Mac
- Enthought eggs for Win32
- Sources
- Web: `http://www.swig.org`
- Documentation: `http://www.swig.org/doc.html`

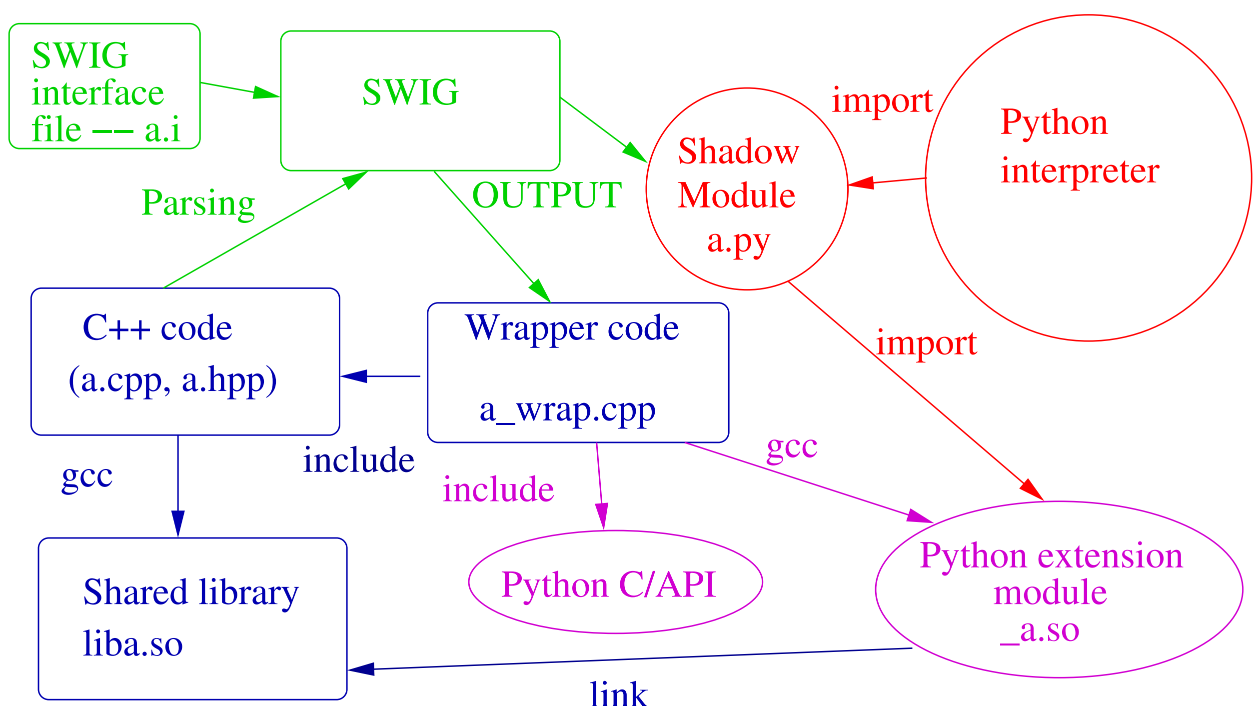
# Using SWIG

1. Build a shared library of the library (`liba.so`)
2. SWIG parses header files using interface file `a.i` to produce:
   - `a_wrap.cpp`: wrapper code for extension module
   - `a.py`: Pure Python module that wraps extension module
3. Build wrapper code to create Python extension: `_a.so`
4. Import `a.py` from Python





# Using SWIG



# Step 0: gcc primer

- *nix only examples follow
- Other platforms and OS options are similar
- Use GCC to typically build object code from C/C++ code
- Use GCC to build all the object code into a shared library
- Typical options:
  - `-c`: Compile object code to object file
  - `-On`: where `n` is $0, 1, \ldots 6$, for optimization, typically `-O2`
  - `-fPIC`: build object code suitably for shared library
  - `-shared`: build shared library
  - `-I<include_path>`: specifies a directory where gcc can find other header files
  - `-L<lib_path>`: specifies a directory where the linker (ld) finds libraries to link to
  - `-l<libname>`: specifies a library to link to





# Step 1: the shared library

```
$ # Generate object file(s).
$ g++ -c a.cpp -fPIC -I<include_path1> \
  -I<include_path2> -o a.o
$ # Build the shared library.
$ g++ -shared a.o -o liba.so -L<lib_path1> \
  -L<lib_path2> -l<libname>
```

- Simple example
- Only Linux for now
- Using Makefiles/SCons here makes life easier
- `-fPIC` needed for shared libraries
- `-I<path>`: optional path to look for header files
- `-L<path>`: optional path to look for libraries
- `-l<libname>`: specifies other libraries to link to

## Step 2: running SWIG

- An interface file `a.i` helps SWIG parse C++ code
- More on interface files later
- `swig -help` gives you more options

```
$ swig -c++ -python <extra-swig-opts> \
  -o a_wrap.cpp a.i
```



Introduction Usage Interface files

## Step 3: building the extension module

```
$ g++ -Wall -O2 -I/usr/include/python2.4 -fPIC \
  -I. -c  -o a_wrap.o a_wrap.cxx
$ g++ -shared -o _a.so a_wrap.o \
  -L. -L<lib_dir> -la
```

- `-la` tells the linker to link the extension to our library
- `_a.so` can be imported from Python
- User should only import `a.py`
- Thats it!

Prabhu Ramachandran Introduction to SWIG

## SWIG wrapping: the big picture

- SWIG Preprocesses all input
- Parses C/C++ declarations, either explicitly specified or in existing headers
- Keeps an internal representation of the code to be wrapped
- Produces wrapper code for the target language
  - Convert basic types to equivalents in Python
  - Everything else is treated as a pointer!
  - SWIG encodes the pointer into some form in Python (like a string with an encoded address) and passes that along to any C/C++ functions
- It is possible to customize each of these aspects using the interface file
- User defined typemaps allow you to control the type conversion code: extremely powerful and quite complex!





## Interface files

- Only absolute basics covered here
- Control how SWIG generates the wrappers
- SWIG C preprocessor: preprocesses all input files
  - Similar but more powerful than C preprocessor
  - File inclusion via `%include` and `import`
  - Conditional compilation
  - Macro expansion
  - Enhanced Macros
  - More details in SWIG documentation
- SWIG is controlled by directives: names preceded by a % symbol

```
%module example
```

## Common directives

- C++-style comments are accepted by SWIG
- `%header`: injected in headers section of wrapper, not parsed

```
%header %{
    ... code in header section ...
%}
// Or in short.
%{ // ...
%}
```

- Module initialization: code that must be called at module initialization

```
%init %{
    // code used at module initialization
    import_array();
%}
```





## Common directives

- Inline: mostly used for helper functions and is parsed by SWIG

```
%inline %{
 Vector *new_Vector() // return new vector
 {return (Vector *) malloc(sizeof(Vector));}
%}
```

## Simple Example C++ code

```
// ------------------------------------
// C++ header: example.hpp
#ifndef _EXAMPLE_H
#define _EXAMPLE_H
float my_constant = 1.01325e5;
long fact(const long n);
#endif
// ------------------------------------
// C++ source: example.cpp
long fact(long n)
{
    if (n == 1) return 1;
    else return n*fact(n-1);
}
```





## Simplest interface file

```
// ------------------------------------
// SWIG interface file: example.i
%module example
%header %{
#include "example.hpp"
%}
// Can also specify the declarations
// here manually, but this works.
%include "example.hpp"
```

## Some comments

- Read SWIG documentation for a detailed exposition
- Easiest to just parse the header using the `%include "header.h"` directive – works for the most part
- If SWIG hasn't parsed a particular structure it treats it like an opaque object
- Any global variables go into the `cvar` variable
- Example: `my_constant` above would be `example.cvar.my_constant`
- Better option is to declare a constant as: `const float some_constant=1.234;`
- Accessible as `example.some_constant`
- Classes and inheritance all work as expected
- Operators that map to clean Python equivalents work
- Namespaces are supported





## Renaming and ignoring declarations

- `%rename` directive renames a function
- `%ignore` ignores a function/struct/class

```
extern void print(char *msg);
extern void import(char *name);
// ------------------------------------
%module example
%rename(my_print) print;
%ignore import;
```

## Input and output parameters

- In C/C++ we use pointers to return multiple values
- Python supports multiple outputs, we'd like to return multiple values from a normal C function
- Also can pass values as pointers
- Like to call this from Python without worrying about pointers
- Use `typemaps.i` and the `INPUT` and `OUTPUT` typemaps





## Input and output parameters ...

```
%{
void getsize(int *xs, int *ys) {
    *xs = 100; *ys = 200;
}
int sub(int *x, int *y) {
    return *x-*y;
}
%}
%include "typemaps.i"
// Apply specifically to arguments
%apply int *OUTPUT {int *xs, int *ys};
void getsize(int *xs, int *ys);
// Declare function suitably.
int  sub(int *INPUT, int *INPUT);
```

## Directors: Overloading virtual C++ functions

- Lets you overload virtuals in Python!
- Creates a subclass that calls the Python method
- Needs to be explicitly requested for

```
%module(directors="1") example
// generate directors for all classes with
// virtual methods
%feature("director");

// generate directors for all
// virtual methods in class Foo
%feature("director") Foo;

// generate a director for just Foo::bar()
%feature("director") Foo::bar;
```





## C++ templates

- Templates are supported but must be instantiated

```
%module example
%{
#include "pair.h"
%}
template<class T1, class T2>
struct pair {
   T1 first;
   T2 second;
   pair();
   pair(const T1&, const T2&);
  ~pair();
};
// Like so:
%template(pairii) pair<int,int>;
```

**Usage in Python**

```
>>> import example as ex
>>> p = ex.pairii(3,4)
>>> p.first
3
>>> p.second
4
```

## std::complex, std::vector etc.

- SWIG library supports wrapping many C++ standard library datatypes
- `%include "std_complex.i"` etc.

```
// SWIG library includes.
%include "std_vector.i"
%include "std_complex.i"
// template instantiation
%template(VectorBool) std::vector<bool>;
%template(VectorDouble) std::vector<double>;
%template(VectorComplexDouble)
 std::vector<std::complex<double> >;
```





## C++ exceptions

- Standard C++ exceptions, trapped in Python and converted to suitable Python exceptions
- Automatically done if the `throw` exception specification is given
- Can also use `%catches` directive
- More options exist: check the SWIG docs

```
%include "std_except.i"
%catches(Error1, Error2) Foo::bar();
class Foo {
public:
    void bar();
    // automatic handling.
    void blah() throw(Error1, Error2, Error3, Error4);
};
```

## Doc strings

```
%module(docstring="This is the example"\
" module's docstring") example
%feature("autodoc", "1");

%feature("docstring") factorial
"long factorial(const long n)
Returns factorial of argument passed. ";
```



Introduction Usage Interface files

## NumPy support

- Can pass in and out `numpy` arrays to C functions
- `numpy.i`
  - defines typemaps for numpy array support
  - available in numpy sources
  - well documented (see `numpy_swig.pdf`)

```
%{
#define SWIG_FILE_WITH_INIT
extern double rms(double* seq, int n);
%}
%include "numpy.i"
%init %{
import_array();
%}
// Note, apply always comes first.
%apply (double* IN_ARRAY1, int DIM1)
       {(double* seq, int n)};
double rms(double* seq, int n);
```

Prabhu Ramachandran Introduction to SWIG

# Summary

- Only a very brief introduction to SWIG
- Please read the docs for more details