Interfaçage de Python avec C++: SWIG

Simplified Wrapper and Interface Generator

Eric Boix

Journée du groupe calcul CNRS

Pourquoi vouloir wrapper (pour Python :)

Soit une librarie C/C++

- Débugger,
- étendre, restreindre, modifier (non intrusif),
- prototyper une utilisation.

Pourquoi vouloir wrapper (pour Python :)

Soit une librarie C/C++

- Débugger,
- étendre, restreindre, modifier (non intrusif),
- prototyper une utilisation.

Dispose d'autres composants sous forme de modules Python

- Prototyper un assemblage (component gluing)
- Ajouter une interface graphique

Pourquoi vouloir wrapper (pour Python :)

Soit une librarie C/C++

- Débugger,
- étendre, restreindre, modifier (non intrusif),
- prototyper une utilisation.

Dispose d'autres composants sous forme de modules Python

- Prototyper un assemblage (component gluing)
- Ajouter une interface graphique

Interfacer C++/Python : étendre python (pas immerger)

Pourquoi vouloir wrapper (pour Python :)

Soit une librarie C/C++

- Débugger,
- étendre, restreindre, modifier (non intrusif),
- prototyper une utilisation.

Dispose d'autres composants sous forme de modules Python

- Prototyper un assemblage (component gluing)
- Ajouter une interface graphique

Interfacer C++/Python : étendre python (pas immerger)
Swig est sous license BSD.

Fonctions de wrappage : conversion des types

example.c

```
int fact(int n) {
    if( n<=1 ) return 1;
    else return n*fact(n-1);
}
```

Fonctions de wrappage : conversion des types

example.c

```
int fact(int n) {
    if( n<=1 ) return 1;
    else return n*fact(n-1);
}
```

```
#include<Python.h>
PyObject *wrap_fact( PyObject *self, PyObject *args) {
    int n, result;
    if( !PyArg_ParseTuple(args, "i:fact",&n))  // INPUT
        return NULL;
    result = fact(n);
    return Py_BuildValue( "i", result);        // OUTPUT
}
```

Initialisation du module : annonce des wrappers

Importation d'un module : annonce des nouvelles methodes

```
bash> python
>>> import example
```

example.so $\longrightarrow$ dlopen() $\longrightarrow$ _init() $\longrightarrow$ initexample()

Initialisation du module : annonce des wrappers

Importation d'un module : annonce des nouvelles methodes

```
bash> python
>>> import example
```

example.so $\longrightarrow$ dlopen() $\longrightarrow$ _init() $\longrightarrow$ initexample()

```
static PyMethodDef exampleMethods[] = {
    { "fact", wrap_fact, 1 },
    { NULL, NULL }
};
void initexample() {
     PyObject *m;
     m = Py_InitModule("example", exampleMethods);
}
```

Module d'extension complet

wrapper.c

```
#include<Python.h>
PyObject *wrap_fact( PyObject *self, PyObject *args) {
    int n, result;
    if( !PyArg_ParseTuple(args, "i:fact",&n))
        return NULL;
    result = fact(n);
    return Py_BuildValue( "i", result);
}
static PyMethodDef exampleMethods[] = {
   { "fact", wrap_fact, 1 }, { NULL, NULL } };
void initexample() {
     PyObject *m;
     m = Py_InitModule("example", exampleMethods);
}
```

Exemple : compilation et utilisation

```
gcc -c example.c
gcc -I/usr/local/include/python2.4 -c wrapper.c
gcc -shared example.o wrapper.o -o example.so
```

Exemple : compilation et utilisation

```
gcc -c example.c
gcc -I/usr/local/include/python2.4 -c wrapper.c
gcc -shared example.o wrapper.o -o example.so

bash> python
>>> import example
>>> example.fact(5)
120
>>>
```

Wrapper du C++ nécessite des outils

Wrapper du C++ nécessite des outils

- Helper classes (parfois intrusif, manuel)
  - PyyCXX
  - Boost-Python (exposé suivant)

Wrapper du C++ nécessite des outils

- Helper classes (parfois intrusif, manuel)
  - PyyCXX
  - Boost-Python (exposé suivant)
- Wrapper : basé sur un parseur C++ (et souvent des helper classes)
  - Py++ (Gcc_xml + Boost-Python)
  - SWIG (internal parser + internal helpers)
  - CableSwig (Gcc_xml + SWIG-patché)

Wrapper du C++ nécessite des outils

- Helper classes (parfois intrusif, manuel)
  - PyyCXX
  - Boost-Python (exposé suivant)
- Wrapper : basé sur un parseur C++ (et souvent des helper classes)
  - Py++ (Gcc_xml + Boost-Python)
  - SWIG (internal parser + internal helpers)
  - CableSwig (Gcc_xml + SWIG-patché)

Mot clefs pour choisir :

- degré d'instrusion,
- degré d'automatisation,
- feature du C++ supportées,
- integration au sein de l'environnement de développement,
- compilo/plateformes supportées...

Notations

example.h

```
extern double Base;
int fact( int n );
```

Notations

example.h

```
extern double Base;
int fact( int n );
```

example.c

```
#include"example.h"  /* Pour tester */
double Base = 4.0;
int fact( int n )
{
    if( n<=1 ) return 1;
    else return n * fact(n-1);
}
```

Fichier d'interface

example.i

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

Fichier d'interface

example.i

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

```
swig -python -o example_wrap.c example.i
[1] Perl/tcl/Ruby/Java/Guile/Ocaml [2] example.py
```

Fichier d'interface

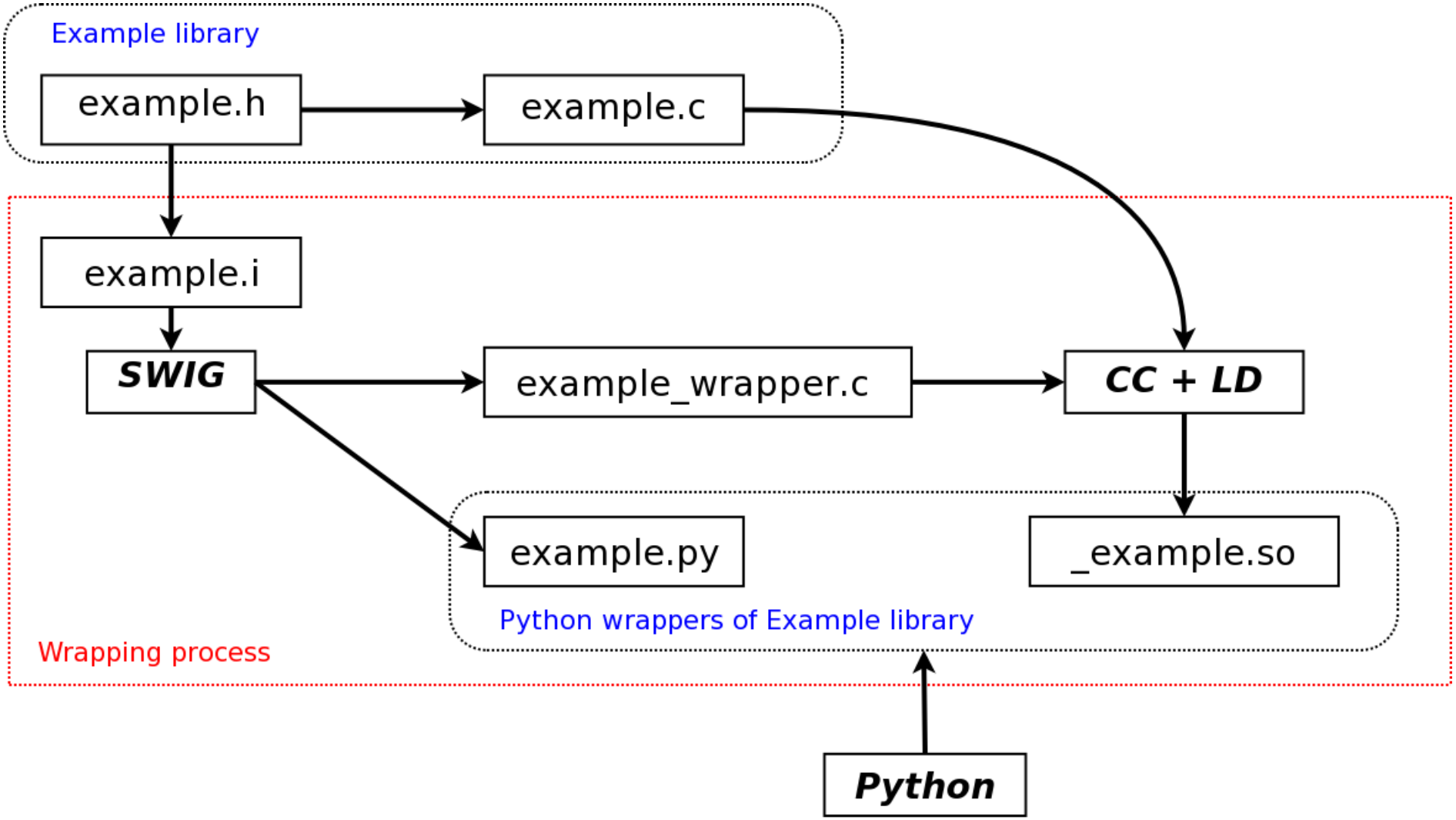
example.i

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

```
swig -python -o example_wrap.c example.i
[1] Perl/tcl/Ruby/Java/Guile/Ocaml [2] example.py

gcc -I/usr/include/python2.4 -c example_wrap.c
gcc -c example.c
gcc -shared example.o example_wrap.o -o _example.so
python -c "import example; print example.fact(5)"
python -c "import example; print example.cvar.Base"
```

Flux de fichiers



Les principales tâches specifiées par SWIG

Essentielement : correspondance des types (Python <—> C), mais aussi des goodies :

Les principales tâches specifiées par SWIG

Essentielement : correspondance des types (Python <—> C), mais aussi des goodies :

- ajout de contraintes [`double sqrt( double NONNEGATIVE )`]
- renomage [`%rename (print) __str__`]
- readonly variables [`%immutable`]
- insertion de code (et wrappage), [`%inline`]
- "Objectification" : du fonctionnel à l'objet Python

Convertion des types de base

Pas de surprise...puisque Python = CPython [java, .Net, OCaml...]

Convertion des types de base

Pas de surprise...puisque Python = CPython [java, .Net, OCaml...]

- int, short, long, char [signed/unsigned] <—> entier Python
- float, double <—> float Python
- char* <—> string Python
- void <—> None
- bool <—> Bolean
- long long, long double : Pas documenté ! ?

Un exemple de typemap

Un typemap est une règle de génération de code attachée à un type du C (au sens typedef, struct) :

```
%typemap(in) int {
    $1 = (int) PyLong_AsLong( $input );
    printf( ``Received and integer : %d\n'', $1 );
}
```

Un peu plus sur les typemaps...

typemaps.i : une bibliotheque de typemap

PairReturn.i

```
%include "typemaps.i"
%apply int *OUTPUT { int *width, int *height }
void getWindowSize( int WinId, int *width, int *height
```

Un peu plus sur les typemaps...

typemaps.i : une bibliotheque de typemap

PairReturn.i

```
%include "typemaps.i"
%apply int *OUTPUT { int *width, int *height }
void getWindowSize( int WinId, int *width, int *height
```

```
>>> w, h = GetWindowSize( MyWindow )
>>>
```

Le modele de pointeur de SWIG

pointer.i

```
%module pointer
%{
#include<stdlib.h>
#include<stdio.h>
%}
void* malloc( unsigned nbytes );
void free( void* );
```

Le modele de pointeur de SWIG

pointer.i

```
%module pointer
%{
#include<stdlib.h>
#include<stdio.h>
%}
void* malloc( unsigned nbytes );
void free( void* );
```

demo.py

```
import pointer
buffer = pointer.malloc( 8192 )
print buffer
pointer.free( buffer )
```

Le modele de pointeur de SWIG

pointer.i

```
%module pointer
%{
#include<stdlib.h>
#include<stdio.h>
%}
void* malloc( unsigned nbytes );
void free( void* );
```

demo.py

```
import pointer
buffer = pointer.malloc( 8192 )
print buffer
pointer.free( buffer )
```

```
<Swig Object of type 'void *' at 0x9b3a9b8>
 9b3a9b8 void p
```

Poussage de patate avec type-checking leger

demoTwo.py

```
import pointer
buffer = pointer.malloc( 8192 )
pointer.free( buffer )
pointer.fclose( buffer) # buffer should be NULL...
```

Poussage de patate avec type-checking leger

demoTwo.py

```
import pointer
buffer = pointer.malloc( 8192 )
pointer.free( buffer )
pointer.fclose( buffer) # buffer should be NULL...
```

```
Traceback (most recent call last):
  File "demoTwo.py", line 4, in ?
    pointer.fclose( buffer)
TypeError: in method 'fclose', argument 1 of type 'FILE *'
```

Array

Les array sont traités comme des pointeurs : pas de conversion en liste ou tuple Python, pas de bound checking...

array.i

```
%module array
%{
double* create_array( int size )
{ return (double*) malloc( (size_t) size ); }
...
 }%
double* create_array( int size );
void spam( double a[1000] );
```

Array

Les array sont traités comme des pointeurs : pas de conversion en liste ou tuple Python, pas de bound checking...

array.i

```
%module array
%{
double* create_array( int size )
{ return (double*) malloc( (size_t) size ); }
...
 }%
double* create_array( int size );
void spam( double a[1000] );
```

```
>>> d = array.create_array(10)
>>> print d
 <Swig Object of type 'double *' at 0x86fef40>
>>> spam( d )
```

Possibilité d´éviter de sérialiser (marshalling)

La définition d'un object complexe...n'est pas toujours nécessaire !

vector.i

```
%module Vector
%{
 ...
 }%
double dot_product( Vector *a, Vector *b );
Vector cross_product( Vector a, Vector b );
Matrix* mat_mul( Matrix *a, Matrix *b );
```

Possibilité d´éviter de sérialiser (marshalling)

La définition d'un object complexe...n'est pas toujours nécessaire !

vector.i

```
%module Vector
%{
...
 }%
double dot_product( Vector *a, Vector *b );
Vector cross_product( Vector a, Vector b );
Matrix* mat_mul( Matrix *a, Matrix *b );
```

Hormis quelques subtilités (SWIG convertit un passage par valeur en passage par reference, y compris pour le retour par valeur)

Possibilité d´éviter de sérialiser (marshalling)

La définition d'un object complexe...n'est pas toujours nécessaire !

vector.i

```
%module Vector
%{
...
 }%
double dot_product( Vector *a, Vector *b );
Vector cross_product( Vector a, Vector b );
Matrix* mat_mul( Matrix *a, Matrix *b );
```

Hormis quelques subtilités (SWIG convertit un passage par valeur en passage par reference, y compris pour le retour par valeur) on peut se limiter au pointeur...

Possibilité d´éviter de sérialiser (marshalling)

La définition d'un object complexe...n'est pas toujours nécessaire !

vector.i

```
%module Vector
%{
...
 }%
double dot_product( Vector *a, Vector *b );
Vector cross_product( Vector a, Vector b );
Matrix* mat_mul( Matrix *a, Matrix *b );
```

Hormis quelques subtilités (SWIG convertit un passage par valeur en passage par reference, y compris pour le retour par valeur) on peut se limiter au pointeur...
...sauf quand la type complexe doit être créé au niveau de Python !

Creation au niveau Python.

Si `Vector est un float[4] alors il faut completer la librairie wrappée avec inline :`

Creation au niveau Python.

Si Vector est un float[4] alors il faut completer la librairie wrappée avec inline :

vector.i

```
double dot_product( Vector *a, Vector *b );
...
%inline %{
Vector* NewVector(float x,float y,float z,float t){
  float* res = (float*)malloc(4*sizeof(float));
  res[0] = x; res[1] = y; res[2] = z; res[3] = t;
  return res;
}  %}
```

Creation au niveau Python.

Si Vector est un float[4] alors il faut completer la librairie wrappée avec inline :

vector.i

```
double dot_product( Vector *a, Vector *b );
...
%inline %{
Vector* NewVector(float x,float y,float z,float t){
  float* res = (float*)malloc(4*sizeof(float));
  res[0] = x; res[1] = y; res[2] = z; res[3] = t;
  return res;
}  %}
```

```
>>> a = NewVector( 0, 1, 2, 3.14159 )
>>> print dot_product( a, a )
```

Creation au niveau Python.

Si Vector est un float[4] alors il faut completer la librairie wrappée avec inline :

vector.i

```
double dot_product( Vector *a, Vector *b );
...
%inline %{
Vector* NewVector(float x,float y,float z,float t){
  float* res = (float*)malloc(4*sizeof(float));
  res[0] = x; res[1] = y; res[2] = z; res[3] = t;
  return res;
}  %}
```

```
>>> a = NewVector( 0, 1, 2, 3.14159 )
>>> print dot_product( a, a )
```

On ne coupe pas toujours au marshalling...

Wrappage d'une classe

Stack.i

```
%module Stack
%inline %{
class Stack {
public:
    Stack();
   ~Stack();
    void push( Object* );
    Object* pop( );
    int depth;
}
%}
```

Wrappage d'une classe

Stack.i

```
%module Stack
%inline %{
class Stack {
public:
    Stack();
   ~Stack();
    void push( Object* );
    Object* pop( );
    int depth;
}
%}
```

```
swig -c++ -python Stack.i
---> Stack_wrap.cxx Stack.py
```

Le resultat : les wrappers en C

Stack_wrap.cxx

```
Stack *wrap_new_Stack() {
    return new Stack;
}
void wrap_delete_Stack( Stack* s ) {
    delete s;
}
void wrap_Stack_push( Stack* s, Object* o ) {
    s->push(o);
}
Object* wrap_Stack_pop( Stack* s ) {
    return s->pop( );
}
```

Les accesseurs

Stack_wrap.cxx

```
int wrap_Stack_depth_get( Stack *s ) {
    return s->depth;
}
int wrap_Stack_depth_set( Stack *s, int d ) {
    s->depth = d;
}
```

Les accesseurs

Stack_wrap.cxx

```
int wrap_Stack_depth_get( Stack *s ) {
    return s->depth;
}
int wrap_Stack_depth_set( Stack *s, int d ) {
    s->depth = d;
}
```

Petit soulevage de capot...

Du wrapper brut de fonderie...

Stack_wrap.cxx

```
SWIGINTERN PyObject
*_wrap_Stack_depth_get(PyObject *SWIGUNUSEDPARM(self), PyObject *args) {
...
  if (!PyArg_ParseTuple(args,
                        (char *)"O:Stack_depth_get",&obj0))
    SWIG_fail;
  res1 = SWIG_ConvertPtr(obj0, &argp1,SWIGTYPE_p_Stack, 0 |  0 );
  if (!SWIG_IsOK(res1)) {
    SWIG_exception_fail(SWIG_ArgError(res1), "argument " "1"" of type '" "Stack *""
  }
  arg1 = reinterpret_cast< Stack * >(argp1);
  result = (int) ((arg1)->depth);
  resultobj = SWIG_From_int(static_cast< int >(result));
  return resultobj;
fail:
  return NULL;
}
```

Du wrapper brut de fonderie...

Stack_wrap.cxx

```
SWIGINTERN PyObject
*_wrap_Stack_depth_get(PyObject *SWIGUNUSEDPARM(self), PyObject *args) {
...
  if (!PyArg_ParseTuple(args,
                        (char *)"O:Stack_depth_get",&obj0))
    SWIG_fail;
  res1 = SWIG_ConvertPtr(obj0, &argp1,SWIGTYPE_p_Stack, 0 |  0 );
  if (!SWIG_IsOK(res1)) {
    SWIG_exception_fail(SWIG_ArgError(res1), "argument " "1"" of type '" "Stack *""
  }
  arg1 = reinterpret_cast< Stack * >(argp1);
  result = (int) ((arg1)->depth);
  resultobj = SWIG_From_int(static_cast< int >(result));
  return resultobj;
fail:
  return NULL;
}
```

Shadow class : seulement pour Python

Stack.py

```
class Stack:
    def __init__( self ):
        self.this = new_Stack()
    def __del__(self):
        delete_Stack( self.this )
    def push(self,o):
        Stack_push( self.this, o )
    def pop(self):
        return Stack_pop( self.this )
    def __getattr__( self, name ):
        if name == 'depth':
            return Stack_depth_get( self.this )
        raise AttributeError,name
```

Utilisation de l'ensemble

```
>>> import Stack
>>> s = Stack()
>>> s.push( "Ben" )
>>> s.push( "George" )
>>> s.pop( )
>>> s.depth
1
```

Shadow class et heritage

Heritage.i

```
%inline %{
class Car {
public:
    enum color_type { white, red, tacky };
    Car( ) { color = white; }
    void paint() { color = red; }
    color_type color;
};
class Limo: public Car {
public:
    Limo() { doors = 18; };
    void paint() { color = tacky; }
    int doors;
};
%}
```

Traduction de l'heritage en Python

```
from Heritage import *

c = Car( )
l = Limo( )

print l.doors      # ---> 18
print l.color      # ---> 0
l.paint()
print l.color      # ---> 2

print isinstance( c, Car )    # ---> True
print isinstance( l, Limo )   # ---> True
print isinstance( l, Car )    # ---> True
```

Traduction de l'heritage en Python

```
from Heritage import *

c = Car( )
l = Limo( )

print l.doors      # ---> 18
print l.color      # ---> 0
l.paint()
print l.color      # ---> 2

print isinstance( c, Car )    # ---> True
print isinstance( l, Limo )   # ---> True
print isinstance( l, Car )    # ---> True
```

Veritable transposition de la hierarchie des classes.

Retour au C : objectifier une librairie C

On peut étendre du C en immitant le mécanisme des shadow classes (ou proxy classes).

Retour au C : objectifier une librairie C

On peut étendre du C en immitant le mécanisme des shadow classes (ou proxy classes).

Image.h

```
struct Image {
    int width;
    int eight;
};
Image* imgcreate(int w, int h);
void imgclear(Image *im, int color);
void imgplot(Image *im, int x, int y);
...
```

Retour au C : objectifier une librairie C

On peut étendre du C en immitant le mécanisme des shadow classes (ou proxy classes).

Image.h

```
struct Image {
    int width;
    int eight;
};
Image* imgcreate(int w, int h);
void imgclear(Image *im, int color);
void imgplot(Image *im, int x, int y);
...
```

Etape 1 : on wrappe (au niveau C) avec SWIG.

Objectifier une librairie C avec Python

Etape 2 : on finit le travail en Python a la mano :

Objectifier une librairie C avec Python

Etape 2 : on finit le travail en Python a la mano :

Image.py

```
from Image import *

Class MyImage:
    def __init__( self, w, h):
        self.this = imgcreate( w, h )
    def clear( self, color ):
        imgclear( self.this, color )
    def plot( self, x, y ):
        imgplot( self.this, color )
...
```

Etendre une classe

On peut aussi étendre du C/C++ au niveau du C/C++

Etendre une classe

On peut aussi étendre du C/C++ au niveau du C/C++

Extend.i

```
struct Image {
    int width;
    int eight;
};

%extend Image {
    void NotInOriginalLib( int parameter ) {
        // code de la methode a ajouter
        ...
    }
};
```

Etendre une classe

On peut aussi étendre du C/C++ au niveau du C/C++

Extend.i

```
struct Image {
    int width;
    int eight;
};

%extend Image {
    void NotInOriginalLib( int parameter ) {
        // code de la methode a ajouter
        ...
    }
};
```

Avantage : permet de bénéficier de l'efficacité du C/C++

Utilisation des templates

Pour les adeptes de la programmation générique :
Il faut obligatoirement instancier (mangler à la mano)

TemplateExample.i

```
%template(intList) List<int>;
class MySpecialList : public List<int> {
...
};
```

Utilisation des templates

Pour les adeptes de la programmation générique :
Il faut obligatoirement instancier (mangler à la mano)

TemplateExample.i

```
%template(intList) List<int>;
class MySpecialList : public List<int> {
...
};
```

...

```
%define TEMPLATE_WRAP(prefix, T...)
%template(prefix ## Foo) Foo<T>;
%template(prefix ## Bar) Bar<T>;
...
%enddef
TEMPLATE_WRAP(String, char *)
TEMPLATE_WRAP(PairStringInt, std::pair<string, int>)
```

En vrac...

- %except permet de remonter de remonter/transposer (ou produire) des exceptions du C++ vers des exceptions de Python

En vrac...

- `%except` permet de remonter de remonter/transposer (ou produire) des exceptions du C++ vers des exceptions de Python
- Swig sait gérer proprement les nested classes (emboîtées) ... si l'emboitement est pas trop profond (CableSwig)

En vrac...

- `%except` permet de remonter de remonter/transposer (ou produire) des exceptions du C++ vers des exceptions de Python
- Swig sait gérer proprement les nested classes (emboîtées) ... si l'emboitement est pas trop profond (CableSwig)

swig parfois mal intégrée dans IDE

Chacun utilise son environnement préféré :

`make`, `GNU-autotools`, `.Net`, `ant`

swig parfois mal intégrée dans IDE

Chacun utilise son environnement préféré :
`make`, `GNU-autotools`, `.Net`, `ant`
Souvent possible parfois pénible [`swig -c++ --> wrap_toto.c`]

swig parfois mal intégrée dans IDE

Chacun utilise son environnement préféré :
`make`, `GNU-autotools`, `.Net`, `ant`
Souvent possible parfois pénible [`swig -c++ --> wrap_toto.c`]

Des solution portables :

- `Distutils` : patcher (2005), pas d'emploi de la STL... et redondance avec la compil de la lib
- `cmake` parfaitement intégrable mais...migration.

Limites actuelles de swig

- support de la STL pas encore documenté.

Limites actuelles de swig

- support de la STL pas encore documenté.
- abscence de Python_d.dll

Limites actuelles de swig

- support de la STL pas encore documenté.
- abscence de Python_d.dll
- Edition de lien incrémentale...

Quelques reflexions générales

- SWIG offre des possibilités technique sans restriction de robustesse :
  - you can shoot your leg off, not mentioning your foot.
  - SWIG's template support provides plenty of opportunities to break the universe.

  Moralité : KISS ou RTFM ou pas mal de temps...

Quelques reflexions générales

- SWIG offre des possibilités technique sans restriction de robustesse :
  - you can shoot your leg off, not mentioning your foot.
  - SWIG's template support provides plenty of opportunities to break the universe.

  Moralité : KISS ou RTFM ou pas mal de temps...
- SWIG n'offre pas de compteur de référence ni de garbage collector pour les allocations faites par le code wrappé... (freaky, leaky)

Quelques reflexions générales

- SWIG offre des possibilités technique sans restriction de robustesse :
  - you can shoot your leg off, not mentioning your foot.
  - SWIG's template support provides plenty of opportunities to break the universe.

  Moralité : KISS ou RTFM ou pas mal de temps...
- SWIG n'offre pas de compteur de référence ni de garbage collector pour les allocations faites par le code wrappé... (freaky, leaky)
- Ne pas céder à une apparente facilité et wrapper avec parsimonie : API design ou plus modestement grow an API.

Références

- Tutorial de David M. Beazley 1998 [http ://www.swig.org/doc.html]
- Tutorial de SWIG
- SWIG-1.3 Development Documentation
  http ://www.swig.org/Doc1.3/index.html