Interfacing C/C++ and Python with SWIG

David M. Beazley

Department of Computer Science
University of Chicago
Chicago, Illinois 60615

beazley@cs.uchicago.edu

Prerequisites

C/C++ programming

- You've written a C program.
- You've written a Makefile.
- You know how to use the compiler and linker.

Python programming

- You've heard of Python.
- You've hopefully written a few Python programs.

Optional, but useful

- Some knowledge of the Python C API.
- C++ programming experience.

Intended Audience

- C/C++ application developers interested in making better programs
- Developers who are adding Python to "legacy" C/C++ code.
- Systems integration (Python as a glue language).

Notes

C/C++ Programming

The good

- High performance.
- Low-level systems programming.
- Available everywhere and reasonably well standardized

The bad

- The compile/debug/nap development cycle.
- Difficulty of extending and modifying.
- Non-interactive.

The ugly

- Writing user-interfaces.
- Writing graphical user-interfaces (worse).
- High level programming.
- Systems integration (gluing components together).

Notes

What Python Brings to C/C++

An interpreted high-level programming environment

- Flexibility.
- Interactivity.
- Scripting.
- Debugging.
- Testing
- Rapid prototyping.

Component gluing

- A common interface can be provided to different C/C++ libraries.
- C/C++ libraries become Python modules.
- Dynamic loading (use only what you need when you need it).

The best of both worlds

- Performance of C
- The power of Python.

Notes

Points to Ponder

"Surely the most powerful stroke for software productivity, reliability, and simplicity has been the progressive use of high-level languages for programming. Most observers credit that development with at least a factor of 5 in productivity, and with concomitant gains in reliability, simplicity, and comprehensibility."

--- **Frederick Brooks**

"The best performance improvement is the transition from the nonworking state to the working state."

--- John Ousterhout

"Less than 10% of the code has to do with the ostensible purpose of the system; the rest deals with input-output, data validation, data structure maintenance, and other housekeeping"

--- Mary Shaw

"Don't keep doing what doesn't work"

--- Anonymous

Notes

Preview

Building Python Modules

- What is an extension module and how do you build one?

SWIG

- Automated construction of Python extension modules.
- Building Python interfaces to C libraries.
- Managing Objects.
- Using library files.
- Customization and advanced features.

Practical Isses

- Working with shared libraries.
- C/C++ coding strategies
- Potential incompatibilities and problems.
- Tips and tricks.

Notes

Python Extension Building

Extending and Embedding Python

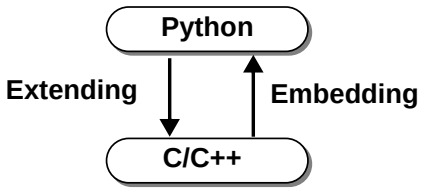
There are two basic methods for integrating C/C++ with Python

- Extension writing.

  Python access to C/C++.

- Embedding

  C/C++ access to the Python interpreter.



We are primarily concerned with “extension writing”.

Notes

Writing Wrapper Functions

"wrapper" functions are needed to access C/C++

- Wrappers serve as a glue layer between languages.
- Need to convert function arguments from Python to C.
- Need to return results in a Python-friendly form.

C Function

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

Wrapper

```
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);
}
```


Notes

The conversion of data between Python and C is performed using two functions :

```
int PyArg_ParseTuple(PyObject *args, char *format, ...)
PyObject *Py_BuildValue(char *format, ...)
```

For each function, the format string contains conversion codes according to the following table :

```
s       =  char *
i       =  int
l       =  long int
h       =  short int
c       =  char
f       =  float
d       =  double
O       =  PyObject *
(items) =  A tuple
|items  =  Optional arguments
```

These functions are used as follows :

```
PyArg_ParseTuple(args,"iid",&a,&b,&c);   // Parse an int,int,double
PyArg_ParseTuple(args,"s|s",&a,&b);      // Parse a string and an optional string
Py_BuildValue("d",value);                // Create a double
Py_BuildValue("(ddd)",a,b,c);            // Create a 3-item tuple of doubles
```

Refer to the Python extending and embedding guide for more details.

Module Initialization

All extension modules need to register wrappers with Python

- An initialization function is called whenever you import an extension module.
- The initialization function registers new methods with the Python interpreter.

A simple initialization function :

```
static PyMethodDef exampleMethods[] = {
         { "fact", wrap_fact, 1 },
         { NULL, NULL }
};

void initexample() {
         PyObject *m;
         m = Py_InitModule("example", exampleMethods);
}
```


Notes

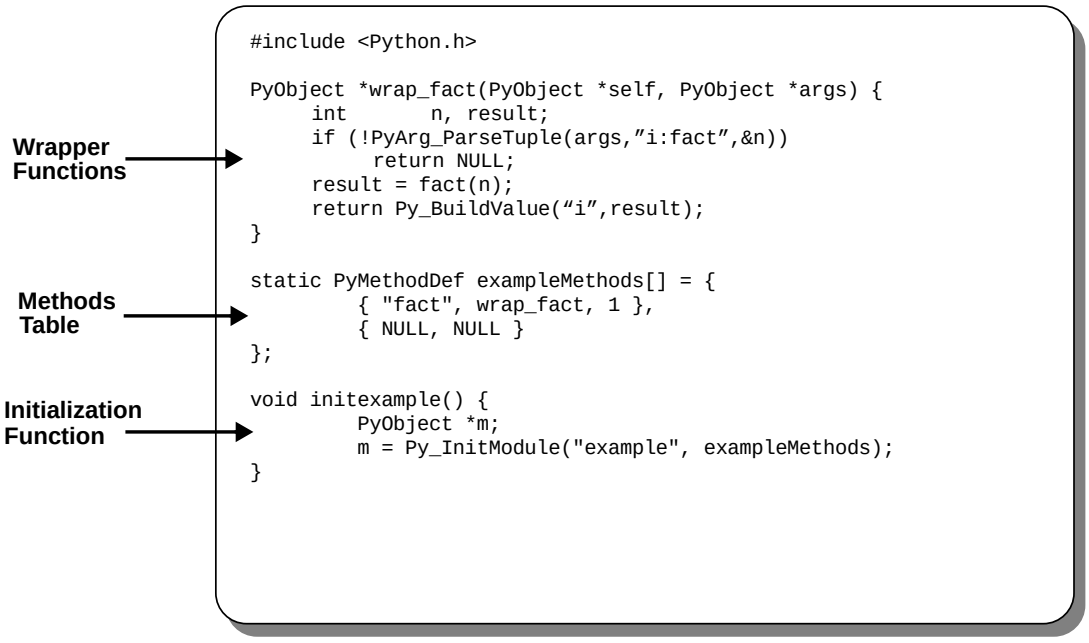
When using C++, the initialization function must be given C linkage. For example :

```
extern "C" void initexample() {
     ...
}
```

On some machines, particularly Windows, it may be necessary to explicitly export the initialization functions. For example,

```
#if defined(__WIN32__)
#   if defined(_MSC_VER)
#       define EXPORT(a,b) __declspec(dllexport) a b
#   else
#       if defined(__BORLANDC__)
#           define EXPORT(a,b) a _export b
#       else
#           define EXPORT(a,b) a b
#       endif
#   endif
#else
#   define EXPORT(a,b) a b
#endif
...
EXPORT(void, initexample) {
     ...
}
```

A Complete Extension Example



Notes

A real extension module might contain dozens or even hundreds of wrapper functions, but the idea is the same.

Compiling A Python Extension

There are two methods

- Dynamic Loading.
- Static linking.

Dynamic Loading

- The extension module is compiled into a shared library or DLL.
- When you type ‘import’, Python loads and initializes your module on the fly.

Static Linking

- The extension module is compiled into the Python core.
- The module will become a new “built-in” module.
- Typing ‘import’ simply initializes the module.

Given the choice, you should try to use dynamic loading

- It’s usually easier.
- It’s surprisingly powerful if used right.

Notes

Most modern operating systems support shared libraries and dynamic loading. To find out more details, view the man-pages for the linker and/or C compiler.

Dynamic Loading

Unfortunately, the build process varies on every machine

- Solaris

```
cc -c -I/usr/local/include/python1.5 \
      -I/usr/local/lib/python1.5/config \
       example.c wrapper.c
ld -G example.o wrapper.o -o examplemodule.so
```

- Linux

```
gcc -fpic -c -I/usr/local/include/python1.5 \
      -I/usr/local/lib/python1.5/config \
      example.c wrapper.c
gcc -shared example.o wrapper.o -o examplemodule.so
```

- Irix

```
cc -c -I/usr/local/include/python1.5 \
      -I/usr/local/lib/python1.5/config \
       example.c wrapper.c
ld -shared example.o wrapper.o -o examplemodule.so
```


Notes

Dynamic Loading (cont...)

- Windows 95/NT (MSVC++)

  Select a DLL project from the AppWizard in Developer Studio. Make sure you add the following directories to the include path

  ```
  python-1.5
  python-1.5\Include
  python-1.5\Pc
  ```

  Link against the Python library. For example :

  ```
  python-1.5\vc40\python15.lib
  ```

Also....

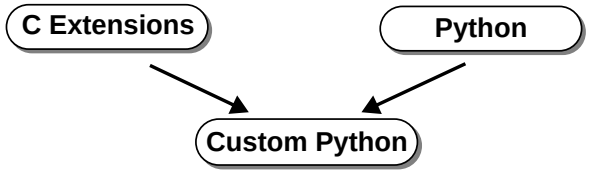
- If your module is named '`example`', make sure you compile it into a file named '`example.so`' or '`examplemodule.so`'.
- You may need to modify the extension to compile properly on all different platforms.
- Not all code can be easily compiled into a shared library (more on that later).

Notes

Static Linking

How it works

• You compile the extension module and link it with the rest of Python to form a new Python executable.



When would you use it?

• When running Python on esoteric machines that don't have shared libraries.

• When building extensions that can't be linked into a shared library.

• If you had a commonly used extension that you wanted to add to the Python core.

Notes

Modifying ‘Setup’ to Add an Extension

To add a new extension module to the Python executable

1. Locate the ‘`Modules`’ directory in the Python source directory.
2. Edit the file ‘`Setup`’ by adding a line such as the following :

```
example example.c wrapper.c
```

Module name **C source files**

3. Execute the script “`makesetup`”
4. Type ‘`make`’ to rebuild the Python executable.

Disadvantages

- Requires the Python source
- May be difficult if you didn’t install Python yourself.
- Somewhat cumbersome during module development and debugging.

Notes

Rebuilding Python by Hand

To manually relink the Python executable (if necessary) :

```
PREFIX        = /usr/local
EXEC_PREFIX   = /usr/local

CC            = cc

PYINCLUDE     = -I$(PREFIX)/include/python1.5 -I$(EXEC_PREFIX)/lib/python1.5/config
PYLIBS        = -L$(EXEC_PREFIX)/lib/python1.5/config \
                  -lModules -lPython -lObjects -lParser
SYSLIBS       = -ldl -lm
PYSRCS        = $(EXEC_PREFIX)/lib/python1.5/config/getpath.c \
                $(EXEC_PREFIX)/lib/python1.5/config/config.c
MAINOBJ       = $(EXEC_PREFIX)/lib/python1.5/config/main.o
PYTHONPATH    = .:$(PREFIX)/lib/python1.5:$(PREFIX)/lib/python1.5/sharedmodules

OBJS          = # Additional object files here

all:
        $(CC) $(PYINCLUDE) -DPYTHONPATH='"$(PYTHONPATH)"' -DPREFIX='"$(PREFIX)"' \
              -DEXEC_PREFIX='"$(EXEC_PREFIX)"' -DHAVE_CONFIG_H $(PYSRCS) \
              $(OBJS) $(MAINOBJ) $(PYLIBS) $(SYSLIBS) -o python
```

Fortunately, there is a somewhat easier way (stay tuned).

Notes

If performing a by-hand build of Python, the file '`config.c`' contains information about the modules contained in the "Setup" script. If needed, you can copy `config.c` and modify it as needed to add your own modules.

The book "Internet Programming with Python", by Watters, van Rossum, and Ahlstrom contains more information about rebuilding Python and the process of adding modules in this manner.

Using The Module

This is the easy part :

```
Python 1.5.1 (#1, May 6 1998)  [GCC 2.7.3]
Copyright 1991-1995 Stichting Mathematisch Centrum,
Amsterdam
>>> import example
>>> example.fact(4)
24
>>>
```

Summary :

- To write a module, you need to write some wrapper functions.
- To build a module, the wrapper code must be compiled into a shared library or staticly linked into the Python executable (this is the tricky part).
- Using the module is easy.
- If all else fails, read the manual (honestly!).

Notes

Wrapping a C Application

The process

- Write a Python wrapper function for every C function you want to access.
- Create Python versions of C constants (not discussed).
- Provide access to C variables, structures, and classes as needed.
- Write an initialization function.
- Compile the whole mess into a Python module.

The problem

- Imagine doing this for a huge library containing hundreds of functions.
- Writing wrappers is extremely tedious and error-prone.
- Consider the problems of frequently changing C code.
- Aren't there better things to be working on?

Notes

Extension Building Tools

Stub Generators (e.g. Modulator)

- Generate wrapper function stubs and provide additional support code.
- You are responsible for filling in the missing pieces and making the module work.

Automated tools (e.g. SWIG, GRAD, bgen, etc...)

- Automatically generate Python interfaces from an interface specification.
- May parse C header files or a specialized interface definition language (IDL).
- Easy to use, but somewhat less flexible than hand-written extensions.

Distributed Objects (e.g. ILU)

- Concerned with sharing data and methods between languages
- Distributed systems, CORBA, COM, ILU, etc...

Extensions to Python itself (e.g. Extension classes, MESS, etc...)

- Aimed at providing a high-level C/C++ API to Python.
- Allow for powerful creation of new Python types, providing integration with C++, etc...

Notes :

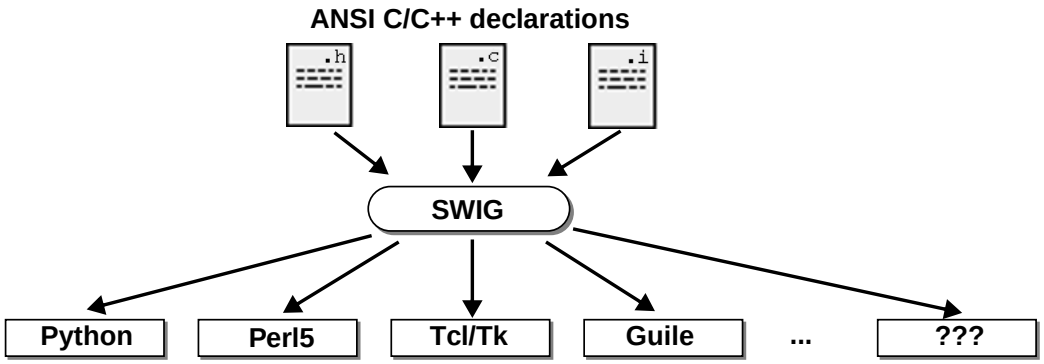
The Python contributed archives contain a wide variety of programming tools. There is no right or wrong way to extend Python--it depends on what kind of problem you're trying to solve. In some cases, you may want to use many of the tools together.

SWIG

An Introduction to SWIG

SWIG (Simplified Wrapper and Interface Generator)

- A compiler that turns ANSI C/C++ declarations into scripting language interfaces.
- Completely automated (produces a fully working Python extension module).
- Language neutral. SWIG can also target Tcl, Perl, Guile, MATLAB, etc...
- Attempts to eliminate the tedium of writing extension modules.



Notes

SWIG Features

Core features

- Parsing of common ANSI C/C++ declarations.
- Support for C structures and C++ classes.
- Comes with a library of useful stuff.
- A wide variety of customization options.
- Language independence (works with Tcl, Perl, MATLAB, and others).
- Extensive documentation.

The SWIG philosophy

- There's more than one way to do it (a.k.a. the Perl philosophy)
- Provide a useful set of primitives.
- Keep it simple, but allow for special cases.
- Allow people to shoot themselves in the foot (if they want to).

Notes

A Simple SWIG Example

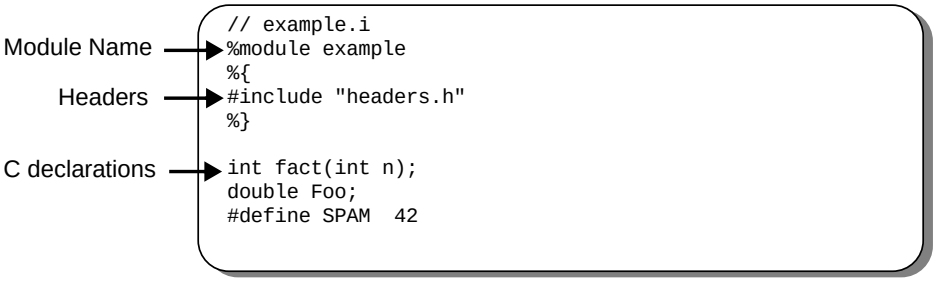
Some C code

```
/* example.c */

double Foo = 7.5;

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

A SWIG interface file



Notes

Scripting interfaces are typically defined in terms of a special “interface file.” This file contains the ANSI C declarations of things you want to access, but also contains SWIG directives (which are always preceded by ’%’). The %module directive specifies the name of the Python extension module. Any code enclosed by %{ ... %} is copied verbatim into the wrapper code generated by SWIG (this is usually used to include header files and other supporting code).

A Simple SWIG Example (cont...)

Building a Python Interface

```
% swig -python example.i
Generating wrappers for Python
% cc -c example.c example_wrap.c \
        -I/usr/local/include/python1.5 \
        -I/usr/local/lib/python1.5/config
% ld -shared example.o example_wrap.o -o examplemodule.so
```

- SWIG produces a file '`example_wrap.c`' that is compiled into a Python module.
- The name of the module and the shared library should match.

Using the module

```
Python 1.5 (#1, May 06 1998)  [GCC 2.7.3]
Copyright 1991-1995 Stichting Mathematisch Centrum,
Amsterdam
>>> import example
>>> example.fact(4)
24
>>> print example.cvar.Foo
7.5
>>> print example.SPAM
42
```


Notes

The process of building a shared library differs on every machine. Refer to earlier slides for more details.

All global variables are accessed through a special object '`cvar`' (for reasons explained shortly).

Troubleshooting tips

- If you get the following error, it usually means that the name of your module and the name of the shared library don't match.

```
>>> import example
Traceback (innermost last):
  File "<stdin>", line 1, in ?
ImportError: dynamic module does not define init function
>>>
```

- If you get the following error, Python may not be able to find your module.

```
>>> import example
Traceback (innermost last):
  File "<stdin>", line 1, in ?
ImportError: No module named example
>>>
```

  To fix this problem, you may need to modify Python's path as follows

```
>>> import sys
>>> sys.path.append("/your/module/path")
>>> import example
```

- The following error usually means your forgot to link everything or there is a missing library.

```
>>> import example
python: can't resolve symbol 'foo'
Traceback (innermost last):
  File "<stdin>", line 1, in ?
ImportError: Unable to resolve symbol
>>>
```

What SWIG Does

Basic C declarations

- C functions become Python functions (or commands).
- C global variables become attributes of a special Python object '`cvar`'.
- C constants become Python variables.

Datatypes

- C built-in datatypes are mapped into the closest Python equivalent.
- `int, long, short` <---> Python integers.
- `float, double` <---> Python floats
- `char, char *` <---> Python strings.
- `void` <---> None
- `long long, long double` ---> Currently unsupported

SWIG tries to create an interface that is a natural extension of the underlying C code.

Notes

- Python integers are represented as '`long`' values. All integers will be cast to and from type `long` when converting between C and Python.
- Python floats are represented as '`double`' values. Single precision floating point values will be cast to type `double` when converting between the languages.
- `long long` and `long double` are unsupported due to the fact that they can not be accurately represented in Python (the values would be truncated).

More on Global Variables

Why does SWIG access global variables through 'cvar'?

"Assignment" in Python

- Variable "assignment" in Python is really just a renaming operation.
- Variables are references to objects.

```
>>> a = [1,2,3]
>>> b = a
>>> b[1] = -10
>>> print a
[1, -10, 3]
```

- A C global variable is not a reference to an object, it is an object.
- To make a long story short, assignment in Python has a meaning that doesn't translate to assignment of C global variables.

Assignment through an object

- C global variables are mapped into the attributes of a special Python object.
- Giving a new value to an attribute changes the value of the C global variable.
- By default, the name of this object is '`cvar`', but the name can be changed.

Notes

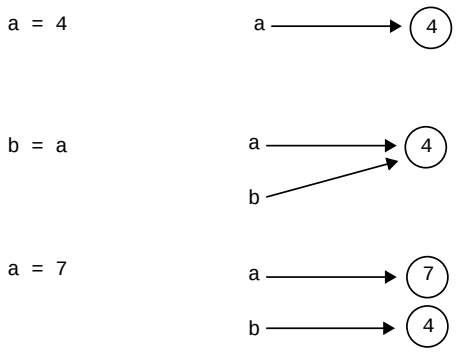
Each SWIG generated module has a special object that is used for accessing C global variables present in the interface. By default the name of this object is '`cvar`' which is short for 'C variables.' If necessary, the name can be changed using the `-globals` option when running SWIG. For example :

```
% swig -python -globals myvar example.i
```

changes the name to '`myvar`' instead.

If a SWIG module contains no global variables, the 'cvar' variable will not be created. Some care is also in order for using multiple SWIG generated modules--if you use the Python '`from module *`' directive, you will get a namespace collision on the value of '`cvar`' (unless you explicitly changed its name as described above).

The assignment model in Python takes some getting used to. Here's a pictorial representation of what's happening.



More on Constants

The following declarations are turned into Python variables

- `#define`
- `const`
- `enum`

Examples :

```
#define ICONST          5
#define FCONST          3.14159
#define SCONST          "hello world"
enum boolean {NO=0, YES=1};
enum months {JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC };
const double PI = 3.141592654;
#define MODE  0x04 | 0x08 | 0x40
```

- The type of a constant is inferred from syntax (unless given explicitly)
- Constant expressions are allowed.
- Values must be defined. For example, '`#define FOO BAR`' does not result in a constant unless BAR has already been defined elsewhere.

Notes

#define is also used by the SWIG preprocessor to define macros and symbols. SWIG only creates a constant if a `#define` directive looks like a constant. For example, the following directives would create constants

```
#define   READ_MODE    1
#define   HAVE_ALLOCA  1
#define   FOOBAR       8.29993
#define   VALUE        4*FOOBAR
```

The following declarations would not result in constants

```
#define USE_PROTOTYPES           // No value given
#define _ANSI_ARGS_(a)    a      // A macro
#define FOO    BAR               // BAR is undefined
```

Pointers

Pointer management is critical!

- Arrays
- Objects
- Most C programs have tons of pointers floating around.

The SWIG type-checked pointer model

- C pointers are handled as opaque objects.
- Encoded with type-information that is used to perform run-time checking.
- Pointers to virtually any C/C++ object can be managed by SWIG.

Advantages of the pointer model

- Conceptually simple.
- Avoids data representation issues (it's not necessary to marshal objects between a Python and C representation).
- Efficient (works with large C objects and is fast).
- It is a good match for most C programs.

Notes

The pointer model allows you to pass pointers to C objects around inside Python scripts, pass pointers to other C functions, and so forth. In many cases this can be done without ever knowing the underlying structure of an object or having to convert C data structures into Python data structures.

An exception to the rule : SWIG does not support pointers to C++ member functions. This is because such pointers can not be properly cast to a pointer of type '`void *`' (the type that SWIG uses internally).

Pointer Example

```
%module example

FILE     *fopen(char *filename, char *mode);
int      fclose(FILE *f);
unsigned fread(void *ptr, unsigned size, unsigned nobj, FILE *);
unsigned fwrite(void *ptr, unsigned size, unsigned nobj, FILE *);

// A memory allocation functions
void    *malloc(unsigned nbytes);
void     free(void *);
```

```
import example
def filecopy(source,target):
     f1 = example.fopen(source,"r")
     f2 = example.fopen(target,"w")
     buffer = example.malloc(8192)
     nbytes = example.fread(buffer,1,8192,f1)
     while nbytes > 0:
          example.fwrite(buffer,1,nbytes,f2)
          nbytes = example.fread(buffer,1,8192,f1)
     example.fclose(f1)
     example.fclose(f2)
     example.free(buffer)
```


Notes

- You can use C pointers in exactly the same manner as in C.
- In the example, we didn't need to know what a `FILE` was to use it (SWIG does not need to know anything about the data a pointer actually points to).
- Like C, you have the power to shoot yourself in the foot. SWIG does nothing to prevent memory leaks, double freeing of memory, passing of NULL pointers, or preventing address violations.

Pointer Encoding and Type Checking

Pointer representation

- Currently represented by Python strings with an address and type-signature.

```
>>> f = example.fopen("test","r")
>>> print f
_f8e40a8_FILE_p
>>> buffer = example.malloc(8192)
>>> print buffer
_1000afe0_void_p
>>>
```

- Pointers are opaque so the precise Python representation doesn't matter much.

Type errors result in Python exceptions

```
>>> example.fclose(buffer)
Traceback (innermost last):
File "<stdin>", line 1, in ?
TypeError: Type error in argument 1 of fclose. Expected _FILE_p.
>>>
```

- Type-checking prevents most of the common errors.
- Has proven to be extremely reliable in practice.

Notes

- The NULL pointer is represented by the string "NULL"
- Python has a special object "CObject" that can be used to hold pointer values. SWIG does not use this object because it does not currently support type-signatures.
- Run-time type-checking is essential for reliable operation because the dynamic nature of Python effectively bypasses all type-checking that would have been performed by the C compiler. The SWIG run-time checker makes up for much of this.
- Future versions of SWIG are likely to change the current pointer representation of strings to an entirely new Python type. This change should not substantially affect the use of SWIG however.

Array Handling

Arrays are pointers

- Same model used in C (the "value" of an array is a pointer to the first element).
- Multidimensional arrays are supported.
- There is no difference between an ordinary pointer and an array.
- However, SWIG does not perform bounds or size checking.
- C arrays are not the same as Python lists or tuples!

```
%module example

double *create_array(int size);
void    spam(double a[10][10][10]);
```

```
>>> d = create_array(1000)
>>> print d
_100f800_double_p
>>> spam(d)
>>>
```


Notes

Pointers and arrays are more-or-less interchangable in SWIG. However, no checks are made to insure that arrays are of the proper size or even initialized properly (if not, you'll probably get a segmentation fault).

It may be useful to re-read the section on arrays in your favorite C programming book---there are subtle differences between arrays and pointers (unfortunately, they are easy to overlook or forget). For example, a pointer of type "double ***" can be accessed as a three-dimensional array, but is not represented in the same way as a three-dimensional array.

Effective use of arrays may require the use of accessor-functions to access individual members (this is described later).

If you plan to do alot of array manipulation, you may want to check out the Numeric Python extension.

Complex Objects

SWIG manipulates all "complex" objects by reference

- The definition of an object is not required.
- Pointers to objects can be freely manipulated.
- Any "unrecognized" datatype is treated as if it were a complex object.

Examples :

```
double dot_product(Vector *a, Vector *b);
FILE *fopen(char *, char *);
Matrix *mat_mul(Matrix *a, Matrix *b);
```


Notes

Whenever SWIG encounters an unknown datatype, it assumes that it is a derived datatype and manipulates it by reference. Unlike the C compiler, SWIG will never generate an error about undefined datatypes. While this may sound strange, it makes it possible for SWIG to build interfaces with a minimal amount of additional information. For example, if SWIG sees a datatype '`Matrix *`', it's obviously a pointer to something (from the syntax). From SWIG's perspective, it doesn't really matter what the pointer is actually pointing to--that is, SWIG doesn't need the definition of `Matrix`.

Passing Objects by Value

What if a program passes complex objects by value?

```
double dot_product(Vector a, Vector b);
```

- SWIG converts pass-by-value arguments into pointers and creates a wrapper equivalent to the following :

```
double wrap_dot_product(Vector *a, Vector *b) {
      return dot_product(*a,*b);
}
```

- This transforms all pass-by-value arguments into pass-by reference.

Is this safe?

- Works fine with C programs.
- Seems to work fine with C++ if you aren't being too clever.

Notes

Trying to implement pass-by-value directly would be extremely difficult---we would be faced with the problem of trying to find a Python representation of C objects (a problem we would rather avoid).

Make sure you tell SWIG about all typedefs. For example,

```
Real spam(Real a);             // Real is unknown.   Use as a pointer
```

versus

```
typedef double Real;
Real spam(Real a);             // Ah. Real is just a 'double'.
```

Return by Value

Return by value is more difficult...

```
Vector cross_product(Vector a, Vector b);
```

- What are we supposed to do with the return value?
- Can't generate a Python representation of it (well, not easily), can't throw it away.
- SWIG is forced to perform a memory allocation and return a pointer.

```
Vector *wrap_cross_product(Vector *a, Vector *b) {
     Vector *result = (Vector *) malloc(sizeof(Vector));
     *result = cross_product(*a,*b);
     return result;
}
```

Isn't this a huge memory leak?

- Yes.
- It is the user's responsibility to free the memory used by the result.
- Better to allow such a function (with a leak), than not at all.

Notes

When SWIG is processing C++ libraries, it uses the default copy constructor instead. For example :

```
Vector *wrap_cross_product(Vector *a, Vector *b) {
     Vector *result = new Vector(cross_product(*a,*b));
     return result;
}
```

Renaming and Restricting

Renaming declarations

- The `%name` directive can be used to change the name of the Python command.

```
%name(output) void print();
```

- Often used to resolve namespace conflicts between C and Python.

Creating read-only variables

- The `%readonly` and `%readwrite` directives can be used to change access permissions to variables.

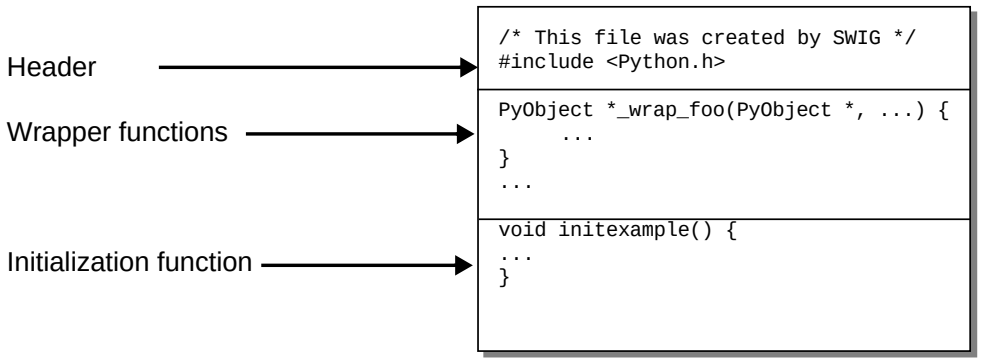
```
double  foo;     // A global variable (read/write)
%readonly
double  bar;     // A global variable (read only)
double  spam;    // (read only)
%readwrite
```

- Read-only mode stays in effect until it is explicitly disabled.

Notes

Code Insertion

The structure of SWIG's output



Four directives are available for inserting code

- `%{ ... %}` inserts code into the header section.
- `%wrapper %{ ... %}` inserts code into the wrapper section.
- `%init %{ ... %}` inserts code into the initialization function.
- `%inline %{ ... %}` inserts code into the header section and "wraps" it.

Notes

These directives insert code verbatim into the output file. This is usually necessary.

The syntax of these directives is loosely derived from YACC parser generators which also use %{,%} to insert supporting code.

Almost all SWIG applications need to insert supporting code into the wrapper output.

Code Insertion Examples

Including the proper header files (extremely common)

```
%module opengl
%{
#include <GL/gl.h>
#include <GL/glu.h>
%}

// Now list declarations
```

Module specific initialization

```
%module matlab
...
// Initialize the module when imported.
%init %{
     eng = engOpen("matlab42");
%}
```


Notes

Inclusion of header files and module specific initialization are two of the most common uses for the code insertion directives.

Helper Functions

Sometimes it is useful to write supporting functions

- Creation and destruction of objects.
- Providing access to arrays.
- Accessing internal pieces of data structures.

```
%module darray
%inline %{
double *new_darray(int size) {
     return (double *) malloc(size*sizeof(double));
}
double darray_get(double *a, int index) {
     return a[index];
}
void darray_set(double *a, int index, double value) {
     a[index] = value;
}
%}

%name(delete_darray) free(void *);
```


Notes

Helper functions can be placed directly inside an interface file by enclosing them in an `%{,%}` block.

Helper functions are commonly used for providing access to various datatypes. For our example above, we would be able to use the functions from Python as follows. For example :

```
from darray import *

# Turn a Python list into a C double array
def createfromlist(l):
     d = new_darray(len(l))
     for i in range(0,len(l)):
          darray_set(d,i,l[i])
     return d

# Print out some elements of an array
def printelements(a, first, last):
     for i in range(first,last):
          print darray_get(a,i)
```

In many cases we may not need to provide Python access, but may need to manufacture objects suitable for passing to other C functions.

Conditional Compilation

Use C preprocessor directives to control SWIG compilation

- The SWIG symbol is defined whenever SWIG is being run.
- Can be used to make mixed SWIG/C header files

```
/* header.h
   A mixed SWIG/C header file */

#ifdef SWIG
%module example
%{
#include "header.h"
%}
#endif

/* C declarations */
...
#ifndef SWIG
/* Don't wrap these declarations. */
#endif
...
```


Notes

SWIG includes an almost complete implementation of the preprocessor that supports `#ifdef`, `#ifndef`, `#if`, `#else`, `#elif`, and `#endif` directives.

File Inclusion

The %include directive

- Includes a file into the current interface file.
- Allows a large interface to be built out of smaller pieces.
- Allows for interface libraries and reuse.

```
%module opengl.i

%include gl.i
%include glu.i
%include aux.i
%include "vis.h"
%include helper.i
```

- File inclusion in SWIG is really like an "import." Files can only be included once and include guards are not required (unlike C header files).

SWIG ignores #include

- Blindly following all includes is probably not what you want.

Notes

Like the C compiler, SWIG library directories can be specified using the -I option. For example :

```
% swig -python -I/home/beazley/SWIG/lib example.i
```

Two other directives, `%extern` and `%import` are also available, but not described in detail. Refer to the SWIG users manual for more information.

Quick Summary

You now know almost everything you need to know

- C declarations are transformed into Python equivalents.
- C datatypes are mapped to an appropriate Python representation.
- Pointers can be manipulated and are type-checked.
- Complex objects are managed by reference.
- SWIG provides special directives for renaming, inserting code, including files, etc...

This forms the foundation for discussing the rest of SWIG.

- Handling of structures, unions, and classes.
- Using the SWIG library.
- Python wrapper classes.
- Customization.
- And more.

Notes

A SWIG Example

Building a Python Interface to OpenGL

OpenGL

- A widely available library/standard for 3D graphics.
- Consists of more than 300 functions and about 500 constants.
- Available on most machines (Mesa is a public domain version).

Interface Building Strategy (in a nutshell)

- Copy the OpenGL header files.
- Modify slightly to make a SWIG interface file.
- Clean up errors and warning messages.
- Write a few support functions.
- Build it.

Why OpenGL?

- It's a significant library that does something real.
- It's available everywhere.
- Can build a simple Python interface fairly quickly.

Notes

The Mesa library is available at

```
http://www.ssec.wisc.edu/~brianp/Mesa.html
```

This example also works with any number of commercial OpenGL implementations.

The Strategy

Locate the OpenGL header files

```
<GL/gl.h>        // Main OpenGL header file
<GL/glu.h>       // Utility functions
<Gl/glaux.h>     // Some auxiliary functions
```

Build the module

- Write a separate interface file for each library.

```
gl.i
glu.i
glaux.i
```

- Combine everything using a master interface file like this

```
// SWIG Interface to OpenGL
%module opengl
%include gl.i
%include glu.i
%include glaux.i
```

- Write a few supporting functions to make the interface work a little better.

Notes

Preparing the Files

opengl.i

```
// OpenGL Interface
%module opengl

%include gl.i
%include glu.i
%include glaux.i
```

gl.i

```
%{
#include <GL/gl.h>
%}
%include "gl.h"
```

glu.i

```
%{
#include <GLU/glu.h>
%}
%include "glu.h"
```

glaux.i

```
%{
#include <GL/glaux.h>
%}
%include "glaux.h"
```

Note : This is only going to be a first attempt.

Notes

A First Attempt

Using SWIG1.2a1 we get the following errors

```
glu.h : Line 231. Error. Function pointer not allowed.
glu.h : Line 271. Error. Function pointer not allowed.
glu.h : Line 354. Error. Function pointer not allowed.
```

Problem : SWIG parser doesn't currently allow function pointers

To fix:

- Copy contents of `glu.h` to `glu.i`
- Edit out offending declarations

```
%{
#include <GL/glu.h>
%}
// Insert glu.h here
...
// void gluQuadricCallback (
//    GLUquadric   *qobj,
//    GLenum        which,
//    void          (CALLBACK *fn)();
```

Also need to fix similar errors in glaux.i

Notes

Second Attempt

No more errors!

- In fact, we can load the module and start executing functions

```
>>> from opengl import *
>>> glClear(GL_DEPTH_BUFFER_BIT)
...
```

- Instant gratification!

But there are other problems

- Many functions are unusable.

```
void glMaterialfv( GLenum face, GLenum pname,
                   const GLfloat *params );
```

- No way to manufacture suitable function arguments

Notes

Helper Functions

Some functions may be difficult to use from Python

```
void glMaterialfv( GLenum face, GLenum pname,
                   const GLfloat *params );
```

- 'params' is supposed to be an array.
- How do we manufacture these arrays in Python and use them?

Write helper functions

```
%inline %{
GLfloat *newfv4(GLfloat a, GLfloat b, GLfloat c, GLfloat d) {
    GLfloat *f = (GLfloat *) malloc(4*sizeof(GLfloat));
    f[0] = a;
    f[1] = b;
    f[2] = c;
    f[3] = d;
    return f;
}
%}
// Create a destructor 'delfv' that is really just 'free'
%name(delfv) void free(void *);
```

- Python lists can also be used as arrays (see section on customization).

Notes

Putting it all together

Final interface file :

```
// opengl.i

%module opengl
%include gl.i
%include glu.i
%include glaux.i
%include help.i
```

The steps (summary) :

- Copy header files.
- Make a few minor changes (only about a dozen in this case).
- Write a few helper functions.
- Compile the module.

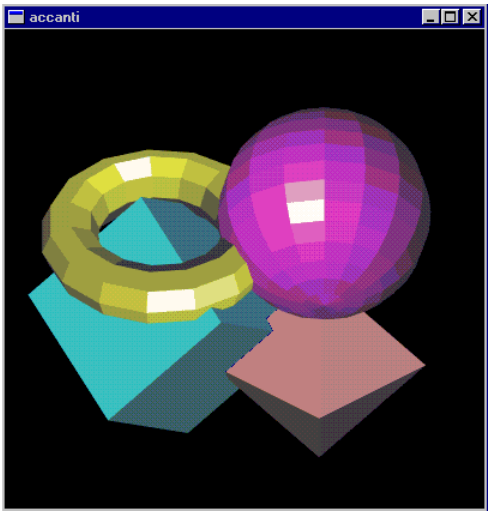
Notes

Python OpenGL Example

```
from opengl import *
...
def displayObjects() :
   torus_diffuse = newfv4(0.7,0.7,0.0,1.0);
   cube_diffuse = newfv4(0.0,0.7,0.7,1.0);
   sphere_diffuse = newfv4(0.7,0.0,0.7,1.0);
   octa_diffuse = newfv4(0.7,0.4,0.4,1.0);
   glPushMatrix();
   glRotatef(30.0, 1.0, 0.0, 0.0);
   glPushMatrix();
   glTranslatef(-0.80, 0.35, 0.0);
   glRotatef(100.0, 1.0, 0.0, 0.0);
   glMaterialfv(GL_FRONT, GL_DIFFUSE,torus_diffuse);
   auxSolidTorus(0.275, 0.85);
   ...

def display() :
   ...

auxInitDisplayMode (AUX_SINGLE | AUX_RGB |
                    AUX_ACCUM | AUX_DEPTH);
auxInitPosition (0, 0, 400, 400);
auxInitWindow ("accanti");
myinit();
reshape(400,400);
display();
...
```



Notes

Summary

Building an interface to a C library is relatively straightfoward

- Can often use header files.
- Might need to make minor changes.
- Write a few helper functions to aid in the process.

Some things to think about

- Wrapping a raw header file might result in an interface that is unusable.
- It is rarely necessary to access everything in a header file from Python.
- SWIG is meant to be fast, but it isn't a substitute for proper planning
- SWIG allows you to "grow" an interface. Start with raw headers and gradually refine the interface until you like it.

Is this the only way?

- No, SWIG provides a variety of customization options.
- Stay tuned.

Objects

Manipulating Objects

The SWIG pointer model (reprise)

- SWIG manages all structures, unions, and classes by reference (i.e. pointers)
- Most C/C++ programs pass objects around as pointers.
- In many cases, writing wrappers and passing opaque pointers is enough.
- However, in some cases you might want more than this.

Issues

- How do you create and destroy C/C++ objects in Python?
- How do you access the internals of an object in Python?
- How do you invoke C++ member functions from Python?
- How do you work with objects in a mixed language environment?

Concerns

- Don’t want to turn Python into C++.
- Don’t want to turn C++ into Python (although this would be an improvement).
- Keep it minimalistic and simple in nature.

Notes

Creating and Destroying Objects

Objects can be created and destroyed by writing special functions :

```
typedef struct {
     double x,y,z;
} Vector;
```

SWIG Interface file

```
%inline %{
Vector *new_Vector(double x, double y, double z) {
     Vector *v = (Vector *) malloc(sizeof(Vector));
     v->x = x; v->y = y; v->z = z;
     return v;
}

void delete_Vector(Vector *v) {
     free(v);
}
%}
```


Notes

Creating and Using Objects in Python

```
>>> v = new_Vector(1, -3, 10)
>>> w = new_Vector(0,-2.5,3)
>>> print v
_1100ef00_Vector_p
>>> print w
_1100ef20_Vector_p
>>> print dot_product(v,w)
37.5
>>> a = cross_product(v,w)
>>> print a
_1100ef80_Vector_p
>>> delete_Vector(v)
>>> delete_Vector(w)
>>> delete_Vector(a)
```

- Special C helper functions are used to construct or destroy objects.
- The model is not radically different than C---we're just manipulating pointers.

Caveat

C/C++ objects created in this manner must be explicitly destroyed. SWIG/Python does not apply reference counting or garbage collection to C/C++ objects.

Notes

While it may be sensible to apply a reference counting scheme to C/C++ objects, this proves to be problematic in practice. There are several factors :

- We often don't know how a "pointer" was manufactured. Unless it was created by `malloc()` or `new`, it would probably be a bad idea to automatically invoke a destructor on it.
- C/C++ programs may use objects in a manner not understood by Python. It would be a bad idea for Python to destroy an object that was still being used inside a C program. Unfortunately, there is no way for Python to know this.
- A C/C++ program may performing its own management (reference counting, smart pointers, etc...). Python wouldn't know about this.

Accessing the Internals of an Object

This is also accomplished using accessor functions

```
%inline %{
double Vector_x_get(Vector *v) {
     return v->x;
}
void Vector_x_set(Vector *v, double val) {
     v->x = val;
}
%}
```

```
>>> v = new_Vector(1,-3,10)
>>> print Vector_x_get(v)
1.0
>>> Vector_x_set(v,7.5)
>>> print Vector_x_get(v)
7.5
>>>
```

- Minimally, you only need to provide access to the “interesting” parts of an object.
- Admittedly crude, but conceptually simple.

Notes

Accessing C++ Member Functions

You guessed it ....

```
class Stack {
public:
     Stack();
     ~Stack();
     void push(Object *);
     Object *pop();
};
```

```
%inline %{
void Stack_push(Stack *s, Object *o) {
     s->push(o);
}
Object *Stack_pop(Stack *s) {
     return s->pop();
}
%}
```

- Basically, we just create ANSI C wrappers around C++ methods.

Notes

Automatic Creation of Accessor Functions

SWIG automatically generates accessor functions if given structure, union or class definitions.

```
%module stack

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

```
Stack  *new_Stack() {
     return new Stack;
}
void    delete_Stack(Stack *s) {
     delete s;
}
void    Stack_push(Stack *s, Object *o) {
     s->push(o);
}
Object *Stack_pop(Stack *s) {
     return s->pop();
}
int     Stack_depth_get(Stack *s) {
     return s->depth;
}
void    Stack_depth_set(Stack *s, int d) {
     s->depth = d;
}
```

- Avoids the tedium of writing the accessor functions yourself.

Notes

Parsing Support for Objects

SWIG provides parsing support for the following

- Basic structure and union definitions.
- Constructors/destructors.
- Member functions.
- Static member functions.
- Static data.
- Enumerations.
- C++ inheritance.

Not currently supported (mostly related to C++)

- Template classes (what is a template in Python?)
- Operator overloading.
- Nested classes.

However, SWIG can work with incomplete definitions

- Just provide the pieces that you want to access.
- SWIG is only concerned with access to objects, not the representation of objects.

Notes

Renaming and Restricting Members

Structure members can be renamed using %name

```
struct Foo {
     %name(spam)    void bar(double);
     %name(status)  int s;
};
```

Access can be restricted using %readonly and %readwrite

```
class Stack {
public:
    Stack();
    ~Stack();
    void push(Object *);
    Object *pop();
    %readonly          // Enable read-only mode
    int depth;
    %readwrite         // Re-enable write access
};
```


Notes

C++ Inheritance and Pointers

SWIG encodes C++ inheritance hierarchies

```
class Shape {
public:
   virtual double area() = 0;
};

class Circle : public Shape {
public:
   Circle(double radius);
  ~Circle();
   double area();
};

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

```
>>> c = new_Circle(7)
>>> s = new_Square(10)
>>> print Square_area(s)
100.0
>>> print Shape_area(s)
100.0
>>> print Shape_area(c)
153.938040046
>>> print Square_area(c)
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: Type error in argument 1 of
Square_area. Expected _Square_p.
>>>
```

- The run-time type checker knows the inheritance hierarchy.
- Type errors will be generated when violations are detected.
- C++ pointers are properly cast when necessary.

Notes

Shadow Classes

Writing a Python wrapper class

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

```
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
```

- Can encapsulate C structures or C++ classes with a Python class
- The Python class serves as a wrapper around the underlying C/C++ object (and is said to "shadow" the object).
- Easily built using pointers and low-level accessor functions.
- Contrast to writing a new Python type in C.

Notes

Automatic Shadow Class Generation

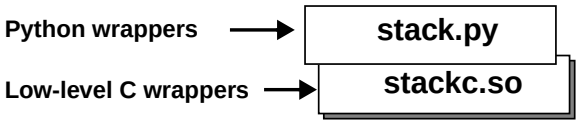
SWIG can automatically generate Python shadow classes

```
% swig -c++ -python -shadow stack.i
```

- When used, SWIG now creates two files :

```
stack.py     #  Python wrappers for the interface
stack_wrap.c #  C wrapper module.
```

Creation of shadow classes results in a layering of two modules



- Typically, only the Python layer is accessed directly.
- The original SWIG interface is still available in the '`stackc`' module.

Shadow classes are just an interface extension

- They utilize pointers and accessor functions.
- No changes to Python are required.

Notes

When using shadow classes, SWIG creates two modules. For example :

```
%module example
...
declarations
...
```

would result into two modules

```
example.py          # The Python wrappers
example_wrap.c      # The C wrappers
```

To build the module, '`example_wrap.c`' file should be compiled and linked into a shared library with the name '`examplecmodule.so`'. Note that an extra 'c' has been appended to the module name.

Now, to use the module, simply use it normally. For example :

```
>>> import example
```

This will load the Python wrappers (and implicitly load the C extension module as well). To use the old C interface, you can load it as follows :

```
>>> import examplec      # Load original C interface.
```

The Anatomy of a Shadow Class

Here's what SWIG really creates :

This class defines the methods available for a generic Stack object (given as a pointer). The constructor for this class simply takes a pointer to an existing object and encapsulates it in a Python class.

This class is used to create a new Stack object. The constructor calls the underlying C/C++ constructor to generate a new object.

```
# This file was created automatically by SWIG.
import stackc
class StackPtr :
    def __init__(self,this):
        self.this = this
        self.thisown = 0
    def __del__(self):
        if self.thisown == 1 :
            stackc.delete_Stack(self.this)
    def push(self,arg0):
        stackc.Stack_push(self.this,arg0)
    def pop(self):
        val = stackc.Stack_pop(self.this)
        return val
    def __setattr__(self,name,value):
        if name == "depth" :
            stackc.Stack_depth_set(self.this,value)
            return
        self.__dict__[name] = value
    def __getattr__(self,name):
        if name == "depth" :
            return stackc.Stack_depth_get(self.this)
        raise AttributeError,name
    def __repr__(self):
        return "<C Stack instance>"

class Stack(StackPtr):
    def __init__(self) :
        self.this = stackc.new_Stack()
        self.thisown = 1
```


Notes

To effectively use shadow classes with real C/C++ programs, you must consider two cases

- The creation of new objects in Python.
- Providing access to objects that have been already created in C/C++.

The use of a pair of Python classes handles both cases. The class `StackPtr` above is used to provide a Python wrapper around an existing Stack object while the class `Stack` is used to create a new Stack object.

One of the reasons for using two classes is to simplify the handling of object construction. If a single class were created, it might be impossible to distinguish the two difficult cases (especially, if the real C/C++ constructor took arguments of its own).

Using a Shadow Class

This is the easy part--they work just like a normal Python class

```
>>> import stack
>>> s = Stack()
>>> s.push("Dave")
>>> s.push("Mike")
>>> s.push("Guido")
>>> s.pop()
Guido
>>> s.depth
2
>>> print s.this
_1008fe8_Stack_p
>>>
```

In practice this works pretty well

- A natural interface to C/C++ structures and classes is provided.
- C++ classes work like Python classes (you can even inherit from them)
- The implementation is relatively simple (it's just a layer over the SWIG pointer mechanism and accessor functions)
- Changes to the Python wrappers are easy to make---they're written in Python

Notes

Nested Objects

Shadow classing even works with nested objects

```
struct Vector {
    double x;
    double y;
    double z;
};

struct Particle {
    Particle();
   ~Particle();
    int      type;
    Vector   r;
    Vector   v;
    Vector   f;
};
```

```
>>> p = Particle()
>>> p.r
<C Vector instance>
>>> p.r.x = 0.0
>>> p.r.y = -7.5
>>> p.r.z = -1.0
>>> print p.r.y
-7.5
>>> p.v = p.r
>>> print p.v.y
-7.5
>>>
```

- SWIG keeps track of various objects and produces appropriate Python code.
- Access to sub-objects works exactly as in C
- Everything is still being manipulated by reference (all operations are being performed directly on the underlying C object without any data copying).

Notes

Managing Object Ownership

Who owns what?

- Objects created by Python are owned by Python (and destroyed by Python)
- Everything else is owned by C/C++.
- The ownership of an object is controlled by the '`thisown`' attribute.

```
self.thisown = 1      # Python owns the object
self.thisown = 0      # C/C++ owns the object.
```

- The owner of an object is responsible for its deletion!

Caveat : sometimes you have to explicitly change the ownership

```
struct Node {
    Node();
   ~Node();
    int    value;
    Node   *next;
};
```

```
# Convert a Python list to a linked list
def listtonode(l):
     n = NodePtr("NULL");
     for i in l:
          m = Node()
          m.value = i
          n.next = m
          n.thisown = 0
          n = m
     return n
```


Notes

If you understand the code example, you can safely say that you understand Python's reference counting mechanism.

In the example, we are saving pointers to objects in the '`next`' field of each data structure. However, consider the use of the variables 'n' and 'm' in the Python code above. As shown, 'n' will be assigned to a new object on each iteration of the loop. Any previous value of 'n' will be destroyed (because there are no longer any Python references to it). Had we not explicitly changed the ownership of the object, this destruction would have also destroyed the original C object. This, in turn, would have created a linked list of invalid pointer values---probably not the effect that you wanted.

When the '`thisown`' variable is set to 0, Python will still destroy 'n' on each iteration of the loop, but this destruction only applies to the Python wrapper class--not the underlying C/C++ object.

Extending Structures and Classes

Object extension : A cool trick for building Python interfaces

- You can provide additional “methods” for use only in Python.
- Debugging.
- Attach functions to C structures (i.e. object-oriented C programming) .

A C structure

```
struct Image {
     int width;
     int height;
   ...
};
```

Some C functions

```
Image *imgcreate(int w, int h);
void imgclear(Image *im, int color);
void imgplot(Image *im,int x,int y,
             int color);
...
```

A Python wrapper class

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


>>> i = Image(400,400)
>>> i.clear(BLACK)
>>> i.plot(200,200,BLUE)
```


Notes

Class Extension with SWIG

The %addmethods directive

```
%module image
struct Image {
     int width;
     int height;
     ...
};
%addmethods Image {
     Image(int w, int h) {
          return imgcreate(w,h);
     }
     void clear(int color) {
          return imgclear(self,color);
     }
     void plot(int x, int y, int color) {
          return imgplot(self,x,y,color);
     }
};
```

- Same syntax as C++.
- Just specify the member functions you would like to have (constructors, destructors, member functions).
- SWIG will combine the added methods with the original structure or class.

Notes

Unlike C++, SWIG uses the variable ‘self’ to hold the original object in added methods. One motivation for this is that class extension is not the same as C++ inheritance nor are the new methods added to any real C++ class. Many C++ compilers would complain about use of the ‘this’ pointer outside of a class so SWIG uses a different name.

Adding Methods (cont...)

Works with both C and C++

- Added methods only affect the Python interface--not the underlying C/C++ code.
- Does not rely upon inheritance or any C++ magic.

How it works (in a nutshell)

- SWIG creates an accessor/helper function, but uses the code you supply.
- The variable 'self' contains a pointer to the corresponding C/C++ object.

```
%addmethods Image {
   ...
   void clear(int color) {
      clear(self,color);
   }
   ...
}
```

```
void Image_clear(Image *self, int color) {
     clear(self,color);
};
```

- If no code is supplied, SWIG assumes that you have already written a function with the required name (methods always have a name like '`Class_method`').
- SWIG treats the added method as if it were part of the original structure/class definition (from Python you will not be able to tell).

Notes

Adding Special Python Methods

%addmethods can be used to add Python specific functions

```
typedef struct {
     double x,y,z;
} Vector;

%addmethods Vector {
...
char *__str__() {
    static char str[256];
    sprintf(str,"[%g, %g, %g]",
            self->x,self->y,self->z);
    return str;
}
};
```

```
>>> v = Vector(2,5.5,9)
>>> print v
[2, 5.5, 9]
>>>
```

- Most of Python's special class methods can be implemented in C/C++ and added to structures or classes.
- Allows construction of fairly powerful Python interfaces.

Notes

The use of a static variable above insures that the '`char *`' returned exists after the function call. Python will make a copy of the returned string when it converts the result to a Python object.

A safer approach would also include some bounds checks on the result string.

Accessing Arrays of Objects

Added methods to the rescue...

```
typedef struct {
     double x,y,z;
} Vector;
...
Vector *varray(int nitems);

%addmethods Vector {
   ...
   Vector *__getitem__(int index) {
        return self+index;
   }
   ...
};
```

```
>>> a = varray(1000)
>>> print a[100]
[0, 0, 0]
>>> for i in range(0,1000):
...     a[i].x = i
>>> print a[500]
[500, 0, 0]
>>>
```

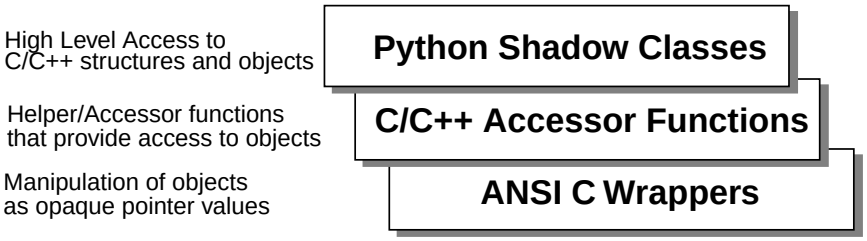
- Accesing arrays of any kind of object is relatively easy.
- Provides natural access (arrays can be manipulated like you would expect).
- Similar tricks can be used for slicing, iteration, and so forth.

Notes

Numeric Python also provides interesting methods for accessing large arrays of numerical data.

Making Sense of Objects (Summary)

SWIG uses a layered approach



All three modes are useful and may be mixed in the same program

- Use opaque pointers when access to an object's internals is unnecessary.
- Use C/C++ accessor functions when occasional access to an object is needed.
- Use Python shadow classes when you want an interface that closely mimics the underlying C/C++ object.

Notes

Interested users might want to compare the SWIG approach to that used in other object systems such as CORBA, ILU, and COM (in fact, some users have used SWIG in conjunction with these systems).

The SWIG Library

The SWIG Library

SWIG is packaged with a standard “library”

- Think of it as the SWIG equivalent of the Python library.

Contents of the library :

- Interface definitions to common C libraries.
- Utility functions (array creation, pointer manipulation, timers, etc...)
- SWIG extensions and customization files.
- Support files (Makefiles, Python scripts, etc...)

Using the library is easy--just use the %include directive.

```
%module example

%include malloc.i
%include pointer.i
%include timers.i
...
```

- Code from the library files is simply inserted into your interface.

Notes

Library Structure

A typical installation

```
/usr/local/lib/swig_lib/
                        /python
                        /tcl
                        /perl5
                        /guile
                        ...
```

- General purpose files (language independent) are placed in the top level
- Language specific extensions are placed in subdirectories. (Python extensions are unavailable when building a Perl module for example).

Setting the library search path (optional)

- Set the environment variable `SWIG_LIB`

```
% setenv SWIG_LIB /usr/local/lib/swig_lib
```

- Use the `-I` option

```
swig -I/usr/local/lib/swig_lib -I/usr/beazley/ifiles \
     interface.i
```


Notes

A Simple Library File

malloc.i

```
// SWIG interface to some memory allocation functions
%module malloc
%{
#include <stdlib.h>
%}
void *malloc(unsigned int size);
void *calloc(unsigned int nobj, unsigned int size);
void *realloc(void *ptr, unsigned int size);
void free(void *ptr);
```

Using the library file

- Copy it to the SWIG library.
- Now, just use '`%include malloc.i`' whenever you want these functions in your interface.

Don't rewrite--build interfaces out of bits and pieces.

Notes

The %module directive found in library files is overridden (or ignored) by any modules that include the file. The functionality of a library file is merely inserted into the module that is being created (i.e. the functions become part of the new module).

Example : The SWIG Pointer Library

%include pointer.i

- Provides high level creation, manipulation, and destruction of common types
- Can create arrays, dereference values, etc...
- The cool part : uses the SWIG type-checker to automatically infer types.

```
%module example
%include pointer.i

void add(double *a, double *b, double *result);
```

```
>>> a = ptrcreate("double",3.5)
>>> b = ptrcreate("double",7.0)
>>> c = ptrcreate("double",0.0)
>>> add(a,b,c)
>>> print ptrvalue(c)
10.5
>>> ptrset(a,-2.0)
>>> print ptrvalue(a)
-2.0
>>> ptrfree(a)
>>> ptrfree(b)
>>> ptrfree(c)
```


Notes

The SWIG pointer library can also perform type-casting, pointer arithmetic, and the equivalent of a run-time 'typedef'. One of the more useful features of the library is its dynamic deferencing operations. For example, `ptrvalue` will return the value of any pointer that is one of the built-in C datatypes (int, long, short, char, float, double, etc...). The type-determination is made dynamically (since all pointers are already encoded with that information).

Library Example : Embedding

%include embed.i

- Includes all of the code needed to build a static version of Python
- The SWIG module is added automatically.
- All builtin Python modules (found in Setup) are also included.
- Compare to the process described earlier.

```
%module example
%include embed.i
...
```

```
% swig -python example.i
% cc -c example_wrap.c -I/usr/local/include/python1.5 \
      -I/usr/local/lib/python1.5/config
% ld example_wrap.o -L/usr/local/lib/python1.5/config \
      -lModules -lPython -lObjects -lParser -ldl -lsocket -lm \
      -o mypython
```


Notes

Library Example : Support Files

Need a Python Makefile in a hurry?

```
% swig -python -co Makefile
Makefile checked out from the SWIG library
%
```

- Copies a preconfigured Python Makefile from the library into the current directory.
- Edit it and you're off and running.

```
# Generated automatically from Makefile.in by configure.
# ---------------------------------------------------------------
# $Header:$
# SWIG Python Makefile
#
# This file can be used to build various Python extensions with SWIG.
# By default this file is set up for dynamic loading, but it can
# be easily customized for static extensions by modifying various
# portions of the file.
#----------------------------------------------------------------

SRCS          =
CXXSRCS       =
... etc ...
```


Notes

In principle any kind of file can be placed in the SWIG library. Think of it as a repository of useful stuff.

Preprocessing

Preprocessing

SWIG contains a modified version of the C preprocessor

- Conditional compilation of interface files.
- Macro expansion (SWIG 1.2).

Caveat : the SWIG preprocessor differs as follows

- `#include` definitions are ignored.
- C/C++ comments are left in the source (these are used for documentation)
- `#define` statements are used to create constants (require special handling).

Note :

SWIG 1.1 and earlier provide limited conditional compilation. SWIG 1.2 (under development) provides full support for macro expansion and most of the other capabilities of a full preprocessor.

Notes

Making Mixed Header/Interface Files

SWIG directives can be placed in C header files

- The SWIG symbol is defined whenever SWIG is running.
- Conditionally compile SWIG directives into a header file when needed.

```
#ifndef _HEADER_H
#define _HEADER_H
#ifdef SWIG
%module example
%{
#include “header.h”
%}
#endif
...

/* Don’t wrap this */
#ifndef SWIG
void foobar(double (*func)(), ...);
#endif
```

This approach makes it a little easier to keep interfaces consistent.

Notes

Working with Macros (SWIG 1.2)

Macro expansion can be used with very complex header files

```
#ifndef HEADER_H
#define HEADER_H
#ifdef SWIG
%module example
%{
#include “header.h”
%}
#endif
#define EXTERN extern
#define _ANSI_ARGS_(a)     a
...
EXTERN void spam _ANSI_ARGS_((int a, double b));
```

- Macros can be used as long as they eventually result in an ANSI C/C++ declaration.
- In extreme cases, one could probably redefine most of SWIG’s syntax (at the expense of really confusing yourself and others).

Notes

SWIG Macros

%define directive can be used for larger macros

```
// Make SWIG wrappers around instantiations of a C++ template class
%{
#include "list.h"
%}
// Define a macro that mirrors a template class definition
%define LIST_TEMPLATE(name,type)
%{
typedef List<type> name;
%}
class name {
public:
     name();
    ~name();
     void append(type);
     int length();
     type get(int n);
};
%enddef

// Now create wrappers around a bunch of different lists
LIST_TEMPLATE(IntList,int)
LIST_TEMPLATE(DoubleList,double)
LIST_TEMPLATE(VectorList, Vector *)
LIST_TEMPLATE(StringList,char *)
```


Notes

While SWIG only supports a subset of C/C++, it often possible to workaround these problems in clever ways. For example, the above code generates wrappers around a few C++ template instantiations using a combination of a 'typedef' and a class definition (which is only given to SWIG).

Advanced SWIG Features

Exception Handling

Python has a nice exception handling mechanism...we should use it.

- Translating C error conditions into Python exceptions.
- Catching C++ exceptions.
- Improving the reliability of our Python modules.

The %except directive

- Allows you to define an application specific exception handler
- Fully configurable (you can do anything you want with it).

```
%except(python) {
     try {
         $function   /* This gets replaced by the real function call */
     }
     catch(RangeError) {
          PyErr_SetString(PyExc_IndexError,"index out-of-bounds");
          return NULL;
     }
}
```

- Exception handling code gets inserted into all of the wrapper functions.

Notes

SWIG Exception Library

SWIG includes a library of generic exception handling functions

- Language independent (works with Python, Tcl, Perl5, etc...)
- Mainly just a set of macros and utility functions.

```
%include exceptions.i
%except(python) {
     try {
          $function
     }
     catch(RangeError) {
          SWIG_exception(SWIG_IndexError,"index out-of-bounds");
     }
}
```

Other things to note

- Exception handling greatly improves the reliability of C/C++ modules.
- However, C/C++ applications need to be written with error handling in mind.
- SWIG can be told to look for errors in any number of ways--as long as there is an error mechanism of some sort in the underlying application.

Notes

SWIG is not limited to C++ exceptions or formal exception handling mechanisms. An exception handling might be something as simple as the following :

```
%except(python) {
     $function
     if (check_error()) {
          char *msg = get_error_msg();
          SWIG_exception(SWIG_RuntimeError,msg);
     }
}
```

where `check_error()` and `get_error_msg()` are C functions to query the state of an application.

Typemaps

Typemaps allow you to change the processing of any datatype

- Handling of input/output values.
- Converting Python objects into C/C++ equivalents (tuples,lists, etc...)
- Telling SWIG to use new Python types.
- Adding constraint handling (the constraint library is really just typemaps).

Very flexible, very powerful

- You can do almost anything with typemaps.
- You can even blow your whole leg off (not to mention your foot).
- Often the topic of discussion on the SWIG mailing list

Caveats

- Requires knowledge of Python's C API to use effectively.
- It's possible to break SWIG in bizarre ways (an interface with typemaps might not even work).
- Impossible to cover in full detail here.

Notes

Typemaps : In a Nutshell

What is a typemap?

- A special processing rule applied to a particular (datatype,name) pair.

```
double spam(int a, int);
```

```
(double,"spam") (int,"a") (int,"")
```

Pattern Matching Rules

- SWIG looks at the input and tries to apply rules using a pattern matching scheme
- Examples :

```
(int,"")         # Matchs all integers
(int,"a")        # Matches only 'int a'
(int *,"")       # Matches 'int *' and arrays.
(int [4],"")     # Matches 'int[4]'
(int [ANY],"")   # Matches any 1-D 'int' array
(int [4][4],"t") # Matches only 'int t[4][4]'
```

- Multiple rules may apply simultaneously
- SWIG always picks the most specific rule.

Notes

The Typemap Library

typemaps.i

- A SWIG library file containing a variety of useful typemaps.
- Handling input/output arguments and other special datatypes.

```
%module example
%include typemaps.i

void add(double *INPUT,double *INPUT, double *OUTPUT);

%apply int OUTPUT { int *width, int *height };
void get_viewport(Image *im, int *width, int *height);
```

```
>>> add(3,4)
7.0
>>> a = get_viewport(im)
>>> print a
[500,500]
>>>
```

- Hmmm. This is much different than the standard pointer model we saw before
- Typemaps allow extensive customization!

Notes

The typemaps.i file contains a number of generally useful typemaps. You should check here before writing a new typemap from scratch.

Writing a New Typemap

Prerequisites

- Need to relatively comfortable with SWIG
- Should be somewhat familar with Python extension writing (and the Python C API)

The %typemap directive

- Used to define new SWIG typemaps at a low level.

```
// Some simple SWIG typemaps
%typemap(python,in) int {
     if (!PyInt_Check($source)) {
        PyErr_SetString(PyExc_TypeError,"Argument $argnum not an integer!");
        return NULL;
     }
     $target = ($type) PyInt_AsLong($source);
}
```

- The code given to a typemap is inserted directly into wrapper functions.
- The variables `$source`,`$target`,`$type`, etc... are filled in with the names of C variables and datatypes.
- Rule of thumb : `$source` is original data. `$target` contains the result after processing.

Notes

How Typemaps Work

```
%typemap(python,in) int {
      ... see previous slide ...
}

int spam(int a, int b);
```

typemap

typemap

```
static PyObject *_wrap_spam(PyObject *self, PyObject *args) {
    PyObject * _resultobj;
    int  _result, _arg0, _arg1;
    PyObject * _obj0 = 0, *_obj1 = 0;
    if(!PyArg_ParseTuple(args,"OO:spam",&_obj0,&_obj1))
        return NULL;
{
        if (!PyInt_Check(_obj0)) {
           PyErr_SetString(PyExc_TypeError,"Argument 1 not an integer!");
           return NULL;
         }
         _arg0 = (int ) PyInt_AsLong(_obj0);
}
{
        if (!PyInt_Check(_obj1)) {
           PyErr_SetString(PyExc_TypeError,"Argument 2 not an integer!");
           return NULL;
         }
         _arg1 = (int ) PyInt_AsLong(_obj1);
}
    _result = (int )spam(_arg0,_arg1);
    _resultobj = Py_BuildValue("i",_result);
    return _resultobj;
}
```


Notes

Typemap Methods

Typemaps can be defined for a variety of purposes

- Function input values (“in”)
- Function output (“out”)
- Default arguments
- Ignored arguments
- Returned arguments.
- Exceptions.
- Constraints.
- Setting/getting of structure members
- Parameter initialization.

The SWIG Users Manual has all the gory details.

Notes

Typemap Applications

Consider our OpenGL example

```
>>> torus_diffuse = newfv4(0.7,0.7,0.0,1.0);
>>> glMaterialfv(GL_FRONT, GL_DIFFUSE,torus_diffuse);
...
>>> delfv4(torus_diffuse)
```

- Needed to manufacture and destroy 4-element arrays using helper functions.

Now a possible typemap implementation

- We define a typemap for converting 4 element tuples to 4 element arrays.
- Rebuild the OpenGL interface with this typemap.

```
>>> torus_diffuse = (0.7,0.7,0.0,1.0)
>>> glMaterialfv(GL_FRONT, GL_DIFFUSE,torus_diffuse)

or simply ...

>>> glMaterialfv(GL_FRONT, GL_DIFFUSE,(0.7,0.7,0.0,1.0))
```

- Yes, that's much nicer now...

Notes

Python Tuple to Array Typemap

```
// Convert a 4 element tuple to a 4 element C array
%typemap(python,in) double[4] (double temp[4]) {
     if (PyTuple_Check($source)) {
         if (!PyArg_ParseTuple($source,"dddd", temp, temp+1, temp+2, temp+3)) {
             PyErr_SetString(PyExc_TypeError,"expected a 4-element tuple of floats");
             return NULL;
         }
         $target = temp;
      } else {
         PyErr_SetString(PyExc_TypeError,"expected a tuple.");
         return NULL;
      }
}
```


Notes

Typemaps : The Bottom Line

Typemaps can be used to customize SWIG

- Changing the handling of specific datatypes.
- Building better interfaces.
- Doing cool things (consider Mark Hammond's Python-COM for instance).

Typemaps can interface with other Python types

- Python lists could be mapped to C arrays.
- You could provide a different representation of C pointers.
- It is possible to use the types of other Python extensions (NumPy, extension classes, etc...).

Some caution is in order

- Typemaps involve writing C/C++ (always risky).
- Understanding the Python C API goes a long way.
- Typemaps may break other parts of SWIG (shadow classes in particular).

Practical Matters

Practical Issues

You've had the grand tour, now what?

- Migrating existing applications to Python.
- Problems and pitfalls in interface generation.
- Working with shared libraries.
- Run-time problems.
- Performance considerations.
- Debugging a Python extension.

Python extension building is only one piece of the puzzle

Notes

Migrating Applications to Python

C/C++ code is usually static and rigid

- Perhaps it's a big monolithic package.
- Control is usually precisely defined.
- Example : parse command line options and do something.

Python/SWIG provides a much more flexible environment

- Can execute any C function in any order.
- Internals are often exposed.
- This is exactly what we want!

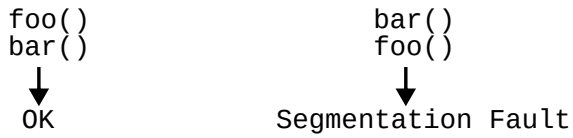
Problem

- Applications may break in mysterious ways.

Notes

Function Execution Dependencies

Functions may implicitly assume prior execution of other functions



- Perhaps foo() was always called before bar() in the original application.

Can add additional state to fix these problems

```
int foo_init = 0;
...
foo() {
    ...
    foo_init = 1;
}
...
bar() {
    if (!foo_init) error("didn't call foo!");
    ...
}
```


Notes

Reentrant Functions

Other functions may only work once

```
bar()    ---> OK
bar()    ---> Error
```

Can fix in the same way

```
bar() {
    static int bar_init = 0;
    if (bar_init) error("already called bar!");
    ...
    bar_init = 1;
}
```


Notes

Namespace Conflicts

C/C++ Namespace collisions

- A C/C++ application may have a namespace conflict with Python's implementation.
- Fortunately this is rare since most Python functions start with 'Py'.
- C/C++ function names may conflict with Python commands.
- C/C++ libraries may have namespace collisions with themselves.

Resolving conflicts with Python built-in commands

- Use the SWIG `%name()` to rename functions.

Resolving conflicts with the Python C implementation

- Change the name of whatever is conflicting (may be able to hide with a macro).

Resolving conflicts between different C libraries

- Tough to fix.
- Dynamic linking may fix the problem.
- Good luck!

Notes

Linking Problems

Extensions usually have to be compiled and linked with the same compiler as Python

- Mismatches may result in dynamic loading errors.
- May just result in a program crash.

Third-party libraries may have problems

- Position independent code often needed for dynamic loading.
- If compiled and linked with a weird compiler, you may be out of luck.

Other components

- SWIG does not provide Python access to generic shared libraries or DLLs.
- Nor do COM components work (look at the Python-COM extension).

Notes

More on Shared Libraries

Shared libraries and C++

- A little more tricky to build than C libraries.
- Require addition runtime support code (default constructors, exceptions, etc...)
- Need to initialize static constructors when loaded.
- Not documented very well.

Rules of thumb when building a dynamic C++ extension

- Try linking the library with the C++ compiler

```
CC -shared example.o example_wrap.o -o examplemodule.so
```

- If that doesn't work, link against the C++ libraries (if you can find them)

```
ld -G example.o example_wrap.o -L/opt/SUNWspro/lib \
      -lC -o examplemodule.so

cc -shared example.o example_wrap.o -lg++ -lstdc++ -lgcc
      -o examplemodule.so
```

- If that still doesn't work, try recompiling Python's main program and relinking the Python executable with the C++ compiler.

Notes

Mixing Shared and Static Libraries

Linking dynamic Python extensions against static libraries is generally a bad idea :

```
/* libspam.a */
static int spam = 7;
int get_spam() {
    return spam;
}
void set_spam(int val) {
    spam = val;
}
```

```
%module foo
...
extern int get_spam();
...
```

```
%module bar
...
extern void set_spam(int);
...
```

- When both Python modules are created, they are linked against `libspam.a`.

What happens :

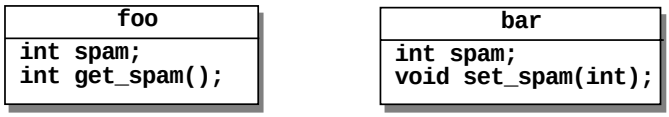
```
>>> import foo
>>> import bar
>>> bar.set_spam(42)
>>> print foo.get_spam()
7

(hmmm... this probably isn't what we expected)
```


Notes

The Static Library Problem

Linking against static libraries results in multiple or incomplete copies of a library



- Neither module contains the complete library (the linker only resolves used symbols).
- Both modules contain a private copy of a variable.

Consider linking against a big library (like OpenGL, etc...)

- Significant internal state is managed by each library.
- Libraries may be resource intensive and have significant interaction with the OS.
- A recipe for disaster.

Solution : use shared libraries

Notes

Using Shared Libraries

If using dynamic loading, use shared libraries

```
/* libspam.so */
static int spam = 7;
int get_spam() {
    return spam;
}
void set_spam(int val) {
    spam = val;
}
```

```
% cc -c spam.c
# Irix
% ld -shared spam.o -o libspam.so
# Solaris
% ld -G spam.o -o libspam.so
# Linux
% cc -shared spam.o -o libspam.so
```

- The process of building a shared library is the same as building a Python extension.

Building and linking Python extensions

- Compile and link normally, but be sure to link against the shared library.

Now it works

```
>>> import foo
>>> import bar
>>> bar.set_spam(42)
>>> foo.get_spam()
42
>>>
```


Notes

More Shared Libraries

Resolving missing libraries

- You may get an error like this :

```
ImportError: Fatal Error : cannot not find 'libspam.so'
```

- The run-time loader is set to look for shared libraries in predefined locations. If your library is located elsewhere, it won't be found.

Solutions

- Set LD_LIBRARY_PATH to include the locations of your libraries

```
% setenv LD_LIBRARY_PATH /home/beazley/app/libs
```

- Link the Python module using an '`rpath`' specifier (better)

```
% ld -shared -rpath /home/beazley/app/libs foo_wrap.o \
       -lspam -o foomodule.so
% ld -G -R /home/beazley/app/libs foo_wrap.o \
       -lspam -o foomodule.so
% gcc -shared -Xlinker -rpath /home/beazley/app/libs \
      foo_wrap.o -lspam -o foomodule.so
```

- Unfortunately, the process varies on every single machine (sigh).

Performance Considerations

Python introduces a performance penalty

- Decoding
- Dispatch
- Execution of wrapper code
- Returning results

These tasks may require thousands of CPU cycles

Rules of thumb

- The performance penalty is small if your C/C++ functions do a lot of work.
- If a function is rarely executed, who cares?
- Don’t write inner loops or perform lots of fine-grained operations in Python.
- Performance critical kernels in C, everything else can be in Python.

From personal experience

- Python inroduces < 1% performance penalty (on number crunching codes).
- Your mileage may vary.

Debugging Dynamic Modules

Suppose one of my Python modules crashes. How do I debug it?

- There is no executable!
- What do you run the debugger on?
- Unfortunately, this is a bigger problem than one might imagine.

My strategy

- Run the debugger on the Python executable itself.
- Run the Python program until it crashes.
- Now use the debugger to find out what's wrong (use as normal).

Caveats

- Your debugger needs to support shared libraries (fortunately most do these days).
- Some debuggers may have trouble loading symbol tables and located source code for shared modules.
- Takes a little practice.

Where to go from here

Topics Not Covered

Modifying SWIG

- SWIG can be extended with new language modules and capabilities.
- Python-COM for example.

Really wild stuff

- Implementing C callback functions in Python.
- Typemaps galore.

SWIG documentation system

- It's being rethought at this time.

Use of other Python extensions

- Modulator
- ILU
- NumPY
- MESS
- Extension classes
- etc....

Notes

SWIG Resources

Web-page

`http://www.swig.org`

FTP-server

`ftp://ftp.swig.org`

Mailing list (for now)

`swig@cs.utah.edu`

To subscribe, send a message 'subscribe swig' to `majordomo@cs.utah.edu`.

Documentation

SWIG comes with about 350 pages of tutorial style documentation (it also supports Tcl and Perl so don't let the size scare you).

Notes