

---

# Using SWIG to Control, Prototype, and Debug C Programs with Python

**Dave Beazley**

Department of Computer Science  
University of Utah  
Salt Lake City, Utah 84112

**beazley@cs.utah.edu**

**June 5, 1996**

---

# Topics

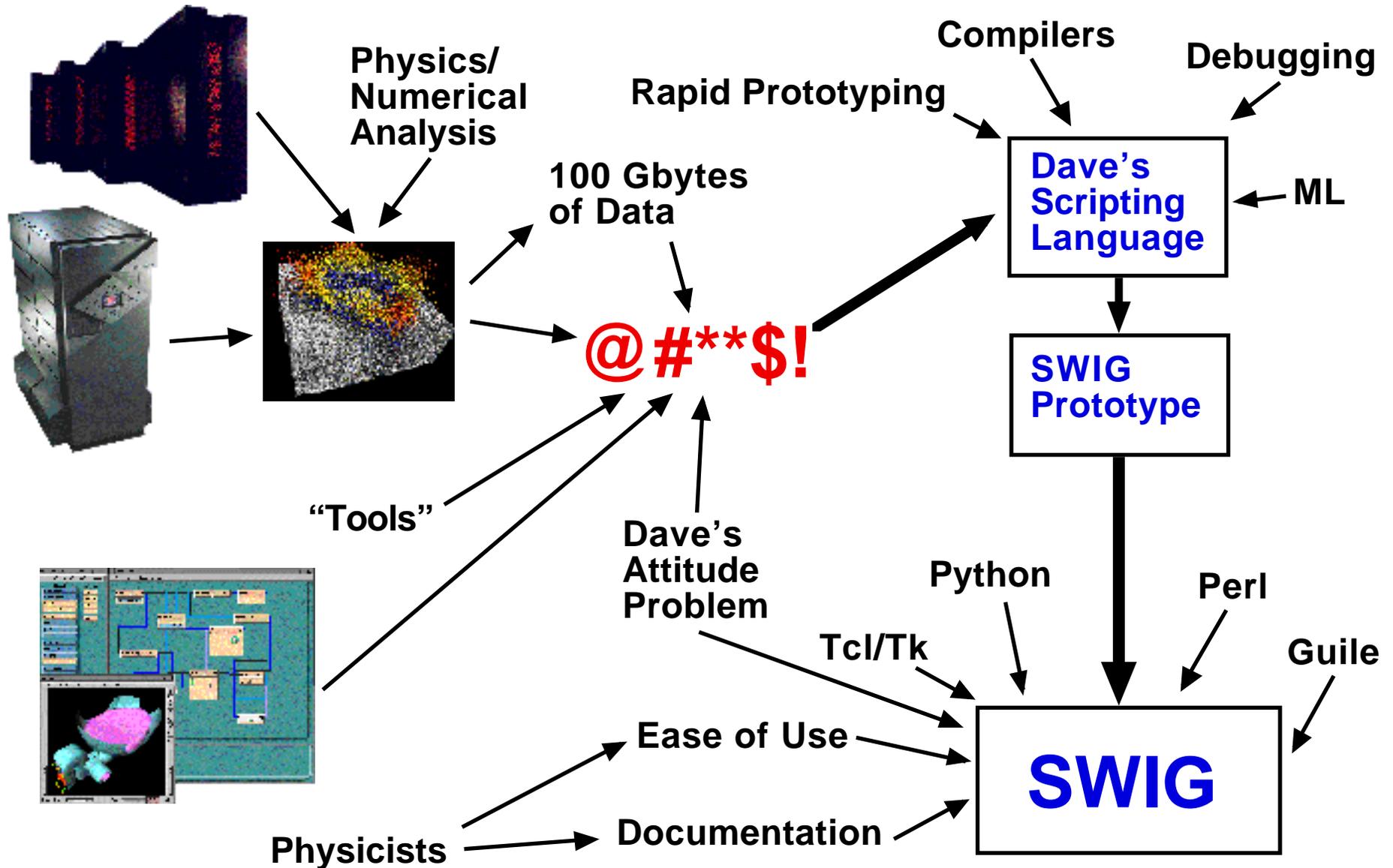
- **What is SWIG?**
- **Extending Python**
- **A Tour of SWIG**
- **Applications**
- **Limitations**
- **Work in progress and future directions**

---

# SWIG (Simplified Wrapper and Interface Generator)

- **Compiler that takes ANSI C/C++ declarations and produces bindings to interpreted languages.**
- **Supports almost all C/C++ datatypes**
- **Binds functions, variables, and constants**
- **C++ classes (including some inheritance)**
- **Run-time type-checking**
- **Supports modules and multiple files**
- **Automatic documentation generation**
- **Currently supports Python, Tcl, Perl4, Perl5, Guile3**
- **Extensible**

# Where am I Coming From?



---

# The Two Language Model

- **Two languages are better than one**
- **C/C++**
  - Performance
  - Hard-core number crunching
  - Portability
  - Complicated stuff
- **Python**
  - Control language
  - Cool things like lists, associative arrays, modules, etc...
  - Interpreted
  - Good extension language
  - Rapid prototyping
  - Debugging
  - User interfaces (tkinter)
- **Use the best features of each language**

---

# Extending Python

Suppose you wanted to add the `getenv()` function to Python.

- Need to write special “wrapper” functions.
- Fortunately, it’s usually not too difficult.
- But, imagine doing this for 200 functions.
- Tedious and error prone.

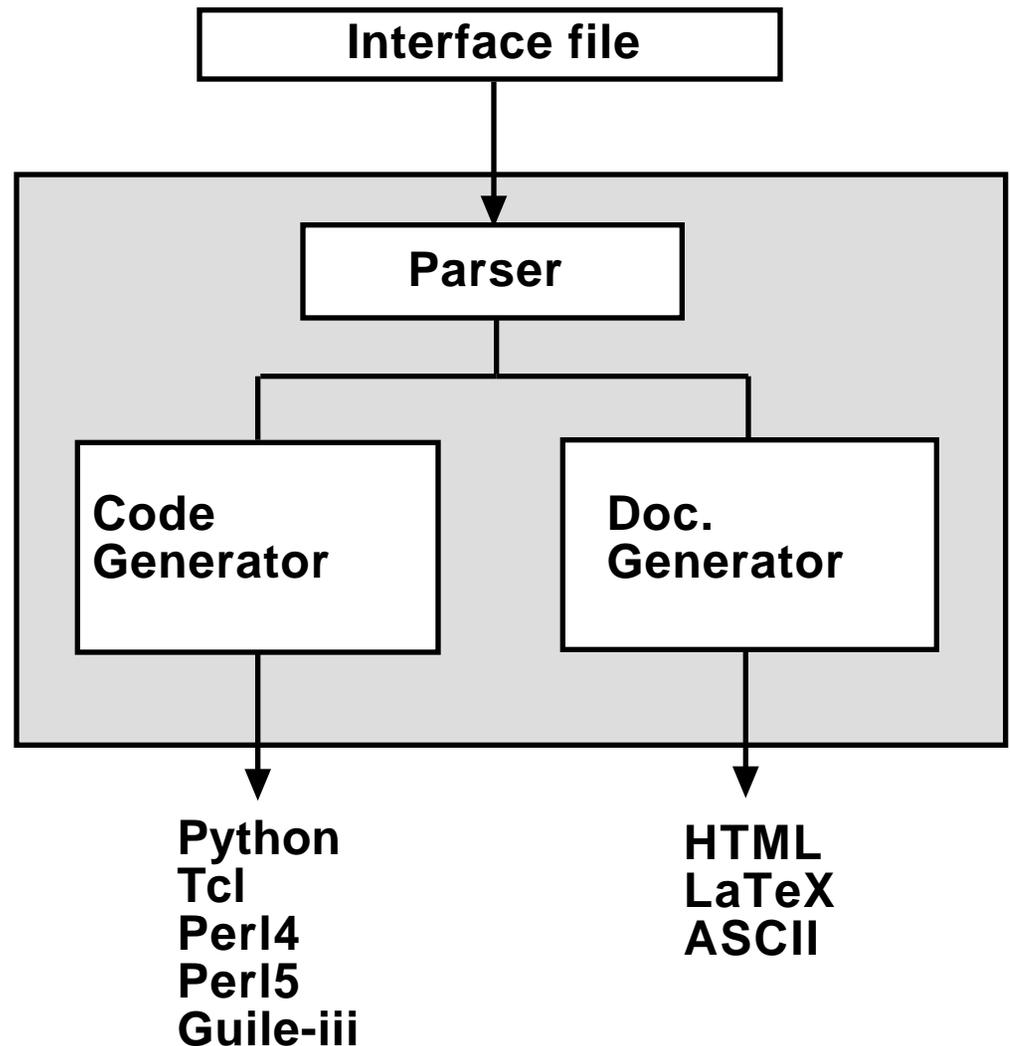
## A Python wrapper function

```
static PyObject *wrap_getenv(  
    PyObject *self,  
    PyObject *args)  
{  
  
    char *result;  
    char *arg0;  
    if (!PyArg_ParseTuple(args, "s", &arg0))  
        return NULL;  
    result = getenv(arg0);  
    return Py_BuildValue("s", result);  
}
```

Procedure is about the same for most other scripting languages (some are easier than others).

# SWIG Overview

- Interface file with ANSI C/C++
- Generic YACC Parser
- Target languages are C++ classes.
- Easy to extend (well mostly).
- Produces a C/C++ source file as output



# A Simple Example

## example.c

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

## example.i (SWIG Interface File)

```
%module example
%{
// Insert headers here
%}
extern int fact(int);
```



```
unix% swig -python example.i
unix% gcc -c example.c example_wrap.c -I/usr/local/include/Py
unix% ld -shared example.o example_wrap.o -o example.so
unix% python1.3
Python1.3 (Apr 12 1996) [GCC 2.5.8]
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>> from example import *
>>> n = fact(6);
>>> print n
720
>>>
```

---

# SWIG Datatypes

- **Built-in types:**

`int, short, long, char, float, double, void`

(integers can also be signed or unsigned).

- **Derived types:**

- Pointers to built-in types.
- Pointers to structures
- Pointers to objects
- Well, almost any kind of pointer actually....

In fact, all derived types are pointers in SWIG (but more on that in a minute).

- **Can remap types using `typedef`**

- **The bottom line :**

- Almost all C/C++ datatypes can be used.
- No pointers to functions (unless hidden via `typedef`).
- Arrays a little weird too...

---

# Adding Functions

- **SWIG uses ANSI C/C++ prototypes :**

```
int          foo(int a, int b);
extern void  print_data(double **);
unsigned int f1(void), f2();
extern int   strcmp(const char *, const char *);
struct tm * localtime(long *t);
double **new_matrix(int n, int m);
Node *get_node(char *);
int     Foo::bar();
```

- **All built-in types and derived types allowed**
- **No type information needed for complex datatypes (ie. Node or struct tm).**
- **Usually pretty easy to use header files.**
- **Parser throws out some clutter (inline code, etc...)**

---

# Linking to C Variables and Constants

- **Access global variables by declaring them**

```
double My_variable;  
extern int flag;  
Node *root;
```

(In python this creates functions such as the following :)

```
double My_variable_get();  
double My_variable_set(double newvalue);
```

- **Create constants with #define, enum, or const**

```
#define MODE 1  
enum colors {red, blue, green, yellow, brown};  
const int FLAGS = 0x04000020;
```

- **Constant expressions allowed**

```
#define STATUS 0x20 | 0x1 | 0x1000  
const double PI_4 = PI/4;
```

---

# Pointers

- All derived types in SWIG are pointers.
- Encoded as strings with type information

Example: `_1008e124_Vector_p`

(Hmmm.... perhaps a carry over from Tcl)

- Checked at run-time for validity

```
>>> n = new_Node()
>>> v = new_Vector(3,5,10)
>>> d = dot(v,n)
Traceback (innermost last):
  File "", line 1, in ?
TypeError: Type error in argument 2 of dot.
Expected _Vector_p.
>>>
```

- Type-checker is savvy to typedef and C++.
- Prevents crashes due to stupid mistakes.

# SWIG and C++

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

class List {
public:
    List();
    ~List();
    int  search(char *item);
    void insert(char *item);
    void remove(char *item);
    char *get(int n);
    int  length;
    static void print(List *l);
};
```



```
List *new_List(void) {
    return new List;
}

void delete_List(List *l) {
    delete l;
}

int List_search(List *l,
                char *item) {
    return l->search(item);
}
...
int List_length_get(List *l) {
    return l->length;
}

int List_length_set(List *l,
                    int value) {
    return (l->length = value);
}

void List_print(List *l) {
    List::print(l);
}
```

- C++ code politely translated into C
- C functions are then wrapped.
- Works independently of the target language.
- Can be fully customized (work in progress).

---

# SWIG and C++ (cont...)

- **Inheritance**

- SWIG supports single public inheritance.
- Virtual functions okay.

- **Use of C++ references usually works**

- **Limitations**

- No operator overloading
- No templates.
- No multiple inheritance
- No function overloading (but you can rename things).

```
                                List();  
%name(ListItem)               List(char *item);
```

- **Thoughts**

- C++ is a complicated beast. Probably pretty hard to support all of it. I've tried to work with a reasonable subset.
- Wasn't my intent to write a full C++ compiler.

---

# Multiple Files and Modules

- **Interfaces can be built from multiple files and libraries.**

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

#include geometry.i
#include file.i
#include graphics.i
#include network.i
#include integration.i
#include mesh.i
```

- **Module system makes it very easy to reuse code in other packages.**
- **SWIG comes with a library of stuff.**
- **Code reuse that works??? (maybe)**

---

# Documentation System

## Documentation is generated automatically from comments

```
%title "My Interface"
%module package
%{
#include "package.h"
%}

%section "Initialization"

void init();
/* Initializes the system */
void haze();
/* Initializes the system and
spawns an initiation ritual */

%section "Graphics"

void LoadMatrix(Matrix *m);
/* Loads viewing transformation matrix */
void ClearBuffer(int color);
/* Clear the frame buffer */

%section "Network"

int open_socket(char *host, int port);
/* Open up a connection with the server */

void disconnect(int fd);
/* Close connection */
```

```
My Interface

1. Initialization

void package.init()
    Initializes the system

void package.haze()
    Initializes the system and spawns
    an initiation ritual.

2. Graphics

void package.LoadMatrix(m)
    Loads viewing transformation
    matrix

void package.ClearBuffer(color)
    Clear the frame buffer

3. Network

int package.open_socket(host, port)
    Open up a connection with the server

void package.disconnect(fd)
    Close connection.
```

---

# Controlling C Programs

- **SWIG requires virtually no modification to C code.**
- **Easy to use with existing applications**
- **Currently used at Los Alamos with the SPaSM Molecular Dynamics code**
  - **~250 functions and variables**
  - **Used on CM-5, T3D, Workstations**
  - **SWIG is hidden in the Makefile and is completely transparent to the users**
- **Simplicity of approach is particularly attractive in research applications.**

---

# Building Python Modules and Classes

- Can build Python modules out of C/C++ libraries.

Example : MATLAB, OpenGL, etc...

- Can encapsulate C++ classes (with a little work)

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

class Tree {
public:
    Tree();
    ~Tree();
    void insert(char *key, char *val);
    char *search(char *key);
    void remove(char *key);
};
```

```
import tree

class Tree:
    def __init__(self):
        self.ptr = new_Tree()
    def __del__(self):
        delete_Tree(self.ptr)
    def insert(self, key, val):
        Tree_insert(self.ptr, key, val)
    def search(self, key):
        v = Tree_search(self.ptr, key)
        return v
    def remove(self, key):
        Tree_remove(self.ptr, key)
```

- Combining various modules usually leads to interesting applications

---

# Prototyping and Debugging

- **SWIG works well if you want to interact with and debug C/C++ code.**
- **Example : OpenGL**

```
%module opengl
%{
%}
#include    gl.i      // All of the OpenGL library
#include    glu.i     // Most of the GLU library
#include    aux.i     // Most of the GL aux library
#include    help.i    // A few functions added to help out
```

**Each include file is just an edited copy of various GL header files.**

**Total development time ~ 20 minutes.**

- **708 Constants**
- **426 Functions**
- **> 8000 lines of wrapper code.**

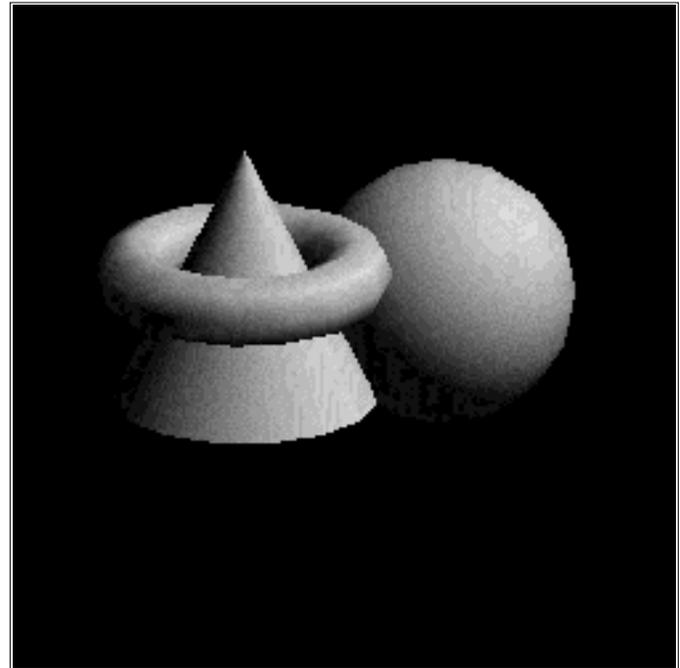
---

# OpenGL Example

```
from OpenGL import *

def myinit ():
    light_ambient = newfv4( 0.0, 0.0, 0.0, 1.0 );
    light_diffuse = newfv4( 1.0, 1.0, 1.0, 1.0 );
    light_specular = newfv4( 1.0, 1.0, 1.0, 1.0 );
    light_position = newfv4( 1.0, 1.0, 1.0, 0.0 );
    glLightfv (GL_LIGHT0, GL_AMBIENT, light_ambient);
    glLightfv (GL_LIGHT0, GL_DIFFUSE, light_diffuse);
    glLightfv (GL_LIGHT0, GL_SPECULAR, light_specular);
    glLightfv (GL_LIGHT0, GL_POSITION, light_position);
    glEnable (GL_LIGHTING);
    glEnable (GL_LIGHT0);
    glDepthFunc(GL_LESS);
    glEnable(GL_DEPTH_TEST);

def display ():
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glPushMatrix ();
    glRotatef (20.0, 1.0, 0.0, 0.0);
    glPushMatrix ();
    glTranslatef (-0.75, 0.5, 0.0);
    glRotatef (90.0, 1.0, 0.0, 0.0);
    auxSolidTorus (0.275, 0.85);
    glPopMatrix ();
    glPushMatrix ();
    glTranslatef (-0.75, -0.5, 0.0);
    glRotatef (270.0, 1.0, 0.0, 0.0);
    auxSolidCone (1.0, 2.0);
    glPopMatrix ();
    glPushMatrix ();
    glTranslatef (0.75, 0.0, -1.0);
    auxSolidSphere (1.0);
    glPopMatrix ();
    glPopMatrix ();
    glFlush ();
```



---

# Current Limitations

- **Lack of variable linking in Python**

Ex.

`double MyVariable ----> MyVariable_get(), MyVariable_set()`

- **Representing pointers as strings is a little weird (But you often don't notice)**
- **Handling of C++ classes not complete. Often have to write a Python wrapper class afterwards**
- **Pointer model is sometimes confusing**
- **No exception model**
- **Numerical representation problems (particularly unsigned integers and longs)**
- **C++ parsing---need I say more?**

---

# A Few Projects Using SWIG

- **Materials Science Simulations (Los Alamos)**
  - SWIG used for nearly a year.
  - System has worked flawlessly.
- **Defibrillation research (Univ. of Utah).**
  - Being used to glue together simulation, mesh generation and visualization code using Tcl/Tk.
- **Synthetic workload generation (Univ. of Utah)**
- **OpenGL widget (Peter-Pike Sloan, Univ. of Utah)**
- **... and many other projects underway.**

---

# Future Directions

- **Release of 1.0 Final (someday)**
- **Continued improvement to Python implementation**
  - Perhaps a new pointer representation
  - Better method for linking with variables
  - Improved C++ support.
  - An exception model???
- **Integration with Numerical Python?**
- **Python for parallel machines?**  
(A little unrelated, but I'd like it)
- **Support for new target languages (as appropriate)**
  - Java (well, maybe)
  - ILU
  - iTcl
- **Support for non-Unix platforms (in progress)**
- **Whatever else comes up...**

---

# Acknowledgments

- **All of the current users who have provided feedback, bug reports, and ideas.**
- **The first users :**  
  
Kurtis Bleeker, Tim Germann, Brad Holian, Peter Lomdahl,  
John Schmidt, Peter-Pike Sloan, Shujia Zhou
- **John Buckman (non-unix platforms and lots of good ideas)**
- **Patrick Tullmann (automatic documentation generation)**
- **The Scientific Computing and Imaging Group**
- **Oh, and these agencies paid some of my salary and provided machines**  
  
**DOE, NSF, NIH**

---

# Advertisement

**The SWIG source code and user manual are available via anonymous FTP:**

`ftp.cs.utah.edu/pub/beazley/SWIG`

**The SWIG homepage:**

`http://www.cs.utah.edu/~beazley/SWIG`

**The SWIG mailing list:**

`swig@cs.utah.edu`

---

# Conclusions

- **SWIG Eliminates a lot of the grungy details of integrating C/C++ with Python**
- **Current system supports a reasonable subset of C and C++.**
- **Two-language model is where it's at.**
- **Many users like the easy of use and straightforward approach.**
- **I developed SWIG for myself---and I find myself using it almost all of the time (but, I'm probably biased).**

---

# Automatic Wrapper Generation

- **Most languages have tools for automatically generating wrapper code.**

ex. ILU, Modulator, Object Tcl, XS, etc...

- **Most tools are specific to a single language**
- **Use of special syntax (or formatting)**
- **Difficult to use.**

## SWIG :

- **Use ANSI C/C++ specifications (independent of target language).**
- **Try to keep it simple, yet flexible.**
- **I hate religious wars....**

---

# Random Thoughts

- **SWIG vs. hand-written modules**
  - SWIG was not really designed to be a generic module builder
- **Performance concerns**
  - You don't use interpreted languages if you want performance.
  - 2 Language Module ==> Write critical stuff in C/C++
- **Security and Reliability**
  - Type-checker eliminates a lot of problems.
  - Can still forge bogus pointer values.
  - C++ is always a little risky (pointer casting problems for instance).
- **Coding Methodology**
  - When used during code development, SWIG encourages modularity.
  - Usually results in more reliable and flexible code.
- **Why all of the Effort?**
  - I want to have a useful and reliable tool.

---

# Building a SWIG Extension

- **Need to supply a new language class** (about a dozen functions).
- **Write a main program like this :**

```
#include <swig.h>
#include <my_python.h>

int main(int argc, char **argv) {
    PYTHON *l;

    l = new PYTHON;
    SWIG_main(argc,argv,l,(Documentation *) 0);
    return 0;
}
```

- **Recompile**

```
unix > g++ my_python.C main.C -lswig -o myswig
```

- **Unfortunately, the code in `my_python.C` is pretty ugly (working on it).**