Tcl Extension Building With SWIG

David M. Beazley

Department of Computer Science
University of Chicago
Chicago, Illinois 60637

beazley@cs.uchicago.edu

Tcl Extension Building

Interfacing Tcl and C/C++ is relatively easy

- Tcl provides a nice C API.
- Can add new commands, variables, objects, widgets, etc...
- Tcl was designed to be extended with compiled code (but you knew this).

However, there are downsides

- Tedious if you have a large library (do you write wrappers for hundreds of functions?)
- Working with structures and C++ classes is difficult.
- Compatibility (the Tcl C API has been known to change from time to time).
- Difficult to manage rapid change (since wrappers must be changed).
- Do you really want to write all of that extension code anyways?

There must be a better way...

Notes

Extension Building Tools

A variety of tools have been developed

- SWIG
- jWrap
- ITcl++
- cpptcl
- Tclobj
- Embedded Tk (ET)
- Object Tcl
- TclObjCommand
- Modular Tcl
- Check the FAQ and contributed archive for more.

Two basic types of tools

- Interface compilers.
- Extensions to the Tcl C API.

All tools are different (try a few and use what you like)

Notes

- SWIG

  `http://www.swig.org`

- jWrap

  `http://www.fridu.com/Html/jWrap.html`

- ITcl++

  `http://www9.informatik.uni-erlangen.de/eng/research/rendering/vision/itcl`

- Embedded Tk (ET)

  `ftp://ftp.neosoft.com/languages/tcl/sorted/devel/et1_5.tar.gz`

- cpptcl

  `http://www.fas.harvard.edu/~darley/EvoXandCpptcl.html`

- Tclobj

  `http://zeus.informatik.uni-frankfurt.de/~fp/Tcl/tclobj`

- TclObjCommand

  `http://ftp.austintx.net/users/jatucker/TclObjectCommand/default.htm`

- Modular Tcl

  `http://www.amath.washington.edu/~lf/software/tcl++`

- Object Tcl

  `http://www.bblink.com/usr/skunk/src/Tools/ObjectTcl-1.1/docs/cover.html`

Why Use an Extension Tool?

Simplicity

- Tools often simplify the construction of complex C/C++ extensions.
- May hide nasty underlying details.
- Automation works well for large packages.

Productivity

- Using a tool is usually faster than writing everything by hand.
- Focus on the problem at hand, not the creation of wrappers.

Better support for rapid change

- Tools are less sensitive to changes in an application.
- Easier to manage new versions of Tcl (e.g., Tcl 7.x to Tcl 8.x).

Allows Tcl to be used in new ways

- As a C/C++ software development tool.
- Testing/debugging.
- Extension tool for end-users (who may not know much about Tcl).

Notes

SWIG

Simplified Wrapper and Interface Generator

- A compiler for the automatic creation of scripting language extensions.
- Supports ANSI C, C++, and Objective-C.
- Creates modules for Tcl 7.x, Tcl 8.x, Perl5, Perl4, Python, and Guile.
- Primary use is building scripting interfaces to existing C/C++ programs.

Availability

- http://www.swig.org
- Supports Unix, Windows-NT, and Macintosh.

Resources

- 340 page user manual (included in the distribution).
- Active mailing list with about 400 subscribers (swig@cs.utah.edu).

Reference

D.M. Beazley, “SWIG : An Easy to Use Tool for Integrating Scripting Languages with C and C++”, in 4th Tcl/Tk Workshop ‘96, Monterey, July 10-13,1996. USENIX, p. 129-139.

Notes

Overview

Topics

- A tour of SWIG.
- Example : Building a Tcl interface to OpenGL
- Objects
- The SWIG Library
- Advanced features
- Limitations and resources.

Prerequisites

- You are an experienced ANSI C programmer.
- You have some familiarity with the Tcl C extension API.
- You have written some Tcl scripts.
- C++ (optional, but useful).

Many of the topics apply to other extension building tools

- SWIG is not the only approach.
- Primary goal is to illustrate the use of a Tcl extension building tool in action.

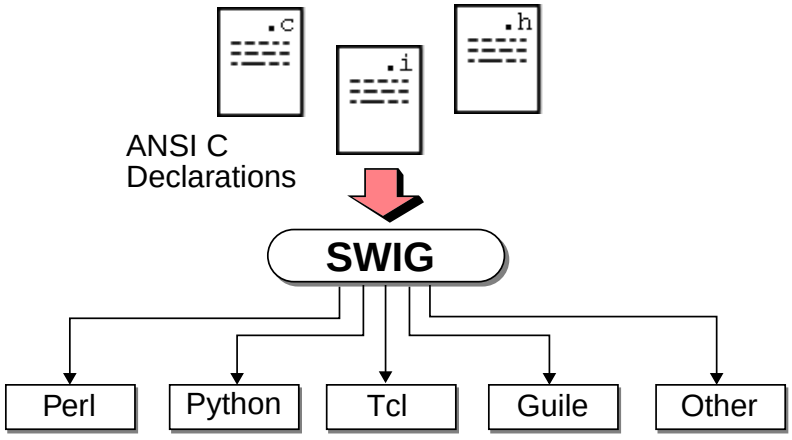
Notes

SWIG Overview

An Introduction

SWIG (Simplified Wrapper and Interface Generator)

- SWIG is a compiler that turns ANSI C/C++ declarations into scripting extensions.
- Completely automated (produces a fully working Tcl extension module).
- Language neutral. SWIG can also target Perl, Python, Guile, MATLAB, and Java.



Notes

SWIG accepts input in the form of ANSI C/C++ declarations that would typically be found in a header file. Input generally comes from three sources--C header files, C source files, and special SWIG “interface files” (which are usually given a .i suffix). In most cases, a combination of different files will be used to build an interface.

ANSI C/C++ syntax was chosen because SWIG was designed to work with existing code. The idea is that you can grab a C header file, tweak it a little bit, and produce a working scripting interface with minimal effort. In other cases, one might create a combined SWIG/C header file that defines everything about your C library (including the Tcl interface).

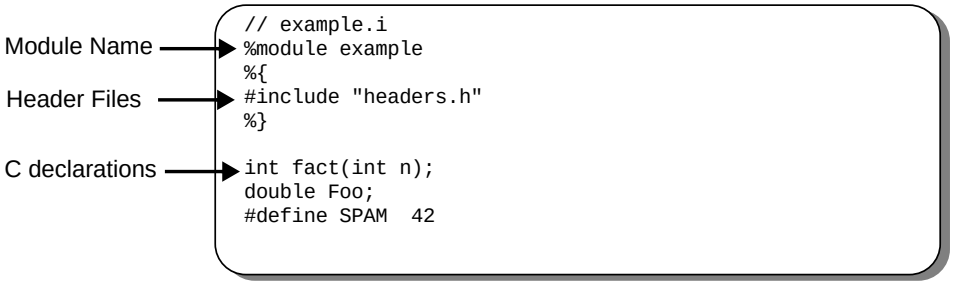
Compare to the interface specification approach used with CORBA IDL or COM.

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

The %module directive specifies the name of the Tcl extension module.

The %{, %} directive is used to insert literal C code into the extension module created by SWIG. This code is simply copied directly into the output file and is not interpreted by the SWIG compiler. Typically, this is used to include header files and any other supporting functions that might be used in the interface.

A Simple SWIG Example (cont...)

Building a Tcl Interface (Linux)

```
% swig -tcl example.i
Generating wrappers for Tcl
% cc -fpic -c example.c example_wrap.c
% cc -shared example.o example_wrap.o -o example.so
```

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

Using the module

```
> tclsh
% load ./example.so
% fact 4
24
% puts $Foo
7.5
% puts $SPAM
42
```

- Can also use packages (described in the SWIG Users Manual).

Notes

Shared libraries

The process of compiling shared libraries varies on every machine. The above example assumes Linux. The following examples show the procedure for Solaris and Irix. Consult the man pages for your C compiler and linker.

- Solaris

```
cc -c example.c wrapper.c
ld -G example.o wrapper.o -o example.so
```

- Irix

```
cc -c example.c wrapper.c
ld -shared example.o wrapper.o -o example.so
```

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.

```
% load ./example.so
couldn't find procedure Example_Init
```

  To fix this problem, make sure the name given with %module matches the name of the shared library.

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

```
% load ./example.so
couldn't load file "./example.so": ./example.so: undefined symbol: fact
```

  To fix this, check the link line to make sure all of the required files and libraries are being used. You need to link the SWIG wrapper file and the original C code together when creating a module. If the original application is packaged as a library, it should be included on the link line when creating the Tcl extension module.

What SWIG Does

Basic C declarations

- C functions become Tcl procedures (or commands).
- C global variables become Tcl variables (if possible--see notes).
- C constants become Tcl variables.

Datatypes

- C built-in datatypes are mapped into the closest Tcl equivalent.
- `int, long, short` <---> Tcl integers.
- `float, double` <---> Tcl float
- `char, char *` <---> Tcl strings.
- `void` <---> Empty string
- `long long, long double` ---> Currently unsupported.
- Tcl Objects are used if the `-tcl8` option is given.

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

Notes

Types of global variables

Global variables are implemented using the Tcl variable linking mechanism. This allows the Tcl interpreter to link to C variables of type `int`, `double`, and `char *`. As a result, C variables of these types appear exactly like normal Tcl variables. Unfortunately, the linking mechanism does not work with all C datatypes. For example,

```
short bar;
```

In this case, SWIG generates a pair of accessor functions

```
short bar_get() { return bar; }
short bar_set(short val) { bar = val; }
```

These functions would then be used in Tcl as follows :

```
% puts [bar_get]
13
% bar_set 6772
```

More on constants

SWIG creates constants from #define, const, and enum declarations. However, #define is also used to define preprocessor macros. Therefore, SWIG only creates a constant from a #define if the value is fully defined. For example, the following declarations create constants :

```
#define  READ_MODE    1
#define  PI           3.14159
#define  PI4          PI/4
```

The following declarations do not result in constants :

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

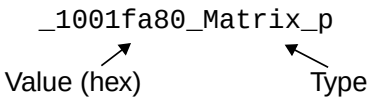
Pointers

Pointer support is critical!

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

SWIG type-checked pointers

- C pointers are mapped to Tcl strings containing the pointer value and type



- Type-signature is used to perform run-time checking.
- Type-violations result in a Tcl error.
- Pointers work exactly like in C except that they can't be dereferenced in Tcl.

Similar to Tcl handles

- Pointer value is a "name" for an underlying C/C++ object.
- SWIG version is more flexible (although perhaps a little more dangerous).

Notes

SWIG allows you to pass pointers to C objects around inside Tcl 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 Tcl data structures.

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-generated extensions use internally).

The NULL pointer is represented by the string "NULL"

Run-time type-checking is essential for reliable operation because the dynamic nature of Tcl effectively bypasses all type-checking that would have been performed by the C compiler.

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 *);
```

```
load ./example.so
proc filecopy {source target} {
     set f1 [fopen $source r]
     set f2 [fopen $target w]
     set buffer [malloc 8192]
     set nbytes [fread $buffer 1 8192 $f1]
     while {$nbytes > 0} {
          fwrite $buffer 1 $nbytes $f2
          set nbytes [fread $buffer 1 8192 $f1]
     }
     fclose $f1
     fclose $f2
     free $buffer
}
```


Notes

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).

More About Pointers

Type errors result in Tcl exceptions

```
% fclose $buffer
Type error in argument 1 of fclose. Expected _FILE_p, received _8062470_void_p
```

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

Interesting features

- Definitions of objects are not required in interface files.
- Almost any C/C++ object can be manipulated with pointers.

Pitfall : Pointers in Tcl are just like pointers in C

- The same rules used in C apply to Tcl.
- Use of malloc/free or new/delete to create and destroy objects.
- Can have dangling pointers, memory leaks, and address violations.
- SWIG assumes you know what you are doing.
- Pointers in Tcl are no less safe than pointers in C.

Notes

Array Handling

Arrays are pointers

- Same as 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 in SWIG.
- SWIG does not perform bounds or size checking.

```
%module example

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

```
set d [create_array 1000]
puts $d
_100f800_double_p
foo $d
```

foo() accepts any "double *"

Notes

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).

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

Objects

SWIG manipulates all objects by reference (i.e., pointers)

```
%module example
%{
#include "vector.h"
%}
Vector *create_vector(double x, double y, double z);
double dot_product(Vector *a, Vector *b);
```

```
% load ./example.so
% set v [create_vector 1.0 2.0 3.0]
% set w [create_vector 4.0 5.0 6.0]
% puts [dot_product $v $w]
32.0
% puts $v
_1008fea8_Vector_p
```

- Can use C/C++ objects without knowing their definition.
- However, can't peer inside objects to view their internal representation.
- SWIG does not complain about undefined datatypes (see note).

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. As a result, the definition of an object is not needed in the interface file.

Passing Objects by Value

What if a program passes 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);
}
```

- 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.

Caveat

- Make sure you tell SWIG about typedef declarations (see notes).

Notes

SWIG converts all undefined types into pointers. As a result, it is important to use typedef correctly. For example,

```
void foo(Real a);             // 'Real' is unknown.  Use as a pointer
```

would get wrapped as follows:

```
void wrap_foo(Real *a) {
     foo(*a);
}
```

In constrast, the following declarations would be wrapped correctly :

```
typedef double Real;
void foo(Real a);             // '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 Tcl 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;
}
```

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.
- Can use the `%inline` directive to add new C functions to an interface.

```
// These functions become part of our Tcl interface
%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;
}
void delete_darray(double *a) {
     free(a);
}
%}
```


Notes

The following Tcl functions show how the above C functions might be used:

```
# Turn a Tcl list into a C double array
proc createfromlist {l} {
     set len [llength $l]
     set d [new_darray $len]
     for {set i 0} { $i < $len } { incr i 1 } {
          darray_set $d $i [lindex $l $i]
     }
     return $d
}

# Print out some elements of an array
proc printelements {a first last} {
     for {set i $first} {$i < $last} {incr i 1 } {
          puts [darray_get $a $i]
     }
}
```

Preprocessing

SWIG has a C preprocessor

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

```
/* header.h
   A mixed SWIG/C header file */
#ifdef SWIG
%module example
%{
#include "header.h"
%}
#endif

#define EXTERN   extern
#ifdef CAN_PROTOTYPE
#define _ANSI_ARGS(a)  a
#else
#define _ANSI_ARGS(a)  ()
#endif
/* C declarations */
EXTERN int foo _ANSI_ARGS((int,double));
...

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


Notes

SWIG1.1 had a partial implementation of a preprocessor that allowed conditional compilation.

SWIG1.2 has a full implementation of a preprocessor that allows conditional compilation and macro expansion.

Preprocessing symbols can be specified on the SWIG command line using the -D option. For example,

```
swig -DDEBUG foo.i
```

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).

Note : SWIG ignores #include statements

- Blindly following all include statements is probably not the behavior you want.

Notes

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

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

Renaming and Restricting

Renaming Declarations

- The %name directive can be used to change the name of the Tcl command

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

- Usually used to resolve namespace conflicts between C and Tcl.

Creating read-only variables

- %readonly and %readwrite directives

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

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

Notes

Miscellaneous Features

Tcl8.0 wrappers

- SWIG will generate wrappers for Tcl objects.
- Run SWIG with the `-tcl8` option.
- Results in better performance.

Namespaces

- Can install all of the wrappers in a Tcl8 namespace.

```
swig -namespace
```

- Namespace name is the same as the module name.
- A C function "bar" in a module "foo" will be wrapped as "foo::bar"

Prefixes

- Can attach a package prefix to functions

```
swig -prefix foo
```

- Not really needed with namespace support.

Notes

Unsupported Features

SWIG is not a full C/C++ parser

- Pointers to functions and arrays are not fully supported.

```
void foo(int (*a)(int, double));          // Error
void bar(int (*b)[50][60]);               // Error
```

- Variable length arguments not supported

```
int fprintf(FILE *f, char *fmt, ...);
```

- Some declarations aren't parsed correctly (a problem in early versions of SWIG)

```
const char *const a;
```

Goal of SWIG is not to parse raw header files!

- ANSI C/C++ syntax used because it is easy to remember and use.
- SWIG interfaces are usually a mix of ANSI C and special SWIG directives.
- Build interfaces by tweaking header files.
- There are workarounds to most parsing problems.

Notes

Quick Summary

You know almost everything you need to know

- C declarations are transformed into Tcl equivalents.
- C datatypes are mapped to an appropriate Tcl representation.
- Pointers can be manipulated and are type-checked.
- Objects are managed by reference.
- SWIG provides special directives for renaming, including files, etc...
- SWIG is not a full C/C++ parser.

This forms the foundation for discussing the rest of SWIG.

- Handling of structures, unions, and classes.
- Using the SWIG library.
- Customization.

A SWIG Example : OpenGL

Building a Tcl Interface to OpenGL

OpenGL

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

Why OpenGL?

- It's real package that does something more than "hello world"
- It's available everywhere.
- An early SWIG user wrapped it in only 10 minutes as his first use of SWIG.

For this example, we'll use

- SWIG1.2a1 on Windows-NT 4.0
- Microsoft Visual C++ 5.0
- Microsoft's implementation of OpenGL
- Tcl 8.0

See notes for Unix information.

Notes

A Unix version of this example can be built using the Mesa library available at

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

Any number of commercial OpenGL implementations should also work.

Interface Building Strategy

Locate the OpenGL header files

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

The plan:

- Write a separate SWIG interface file for each header.

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

- Combine everything using an interface file similar to 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

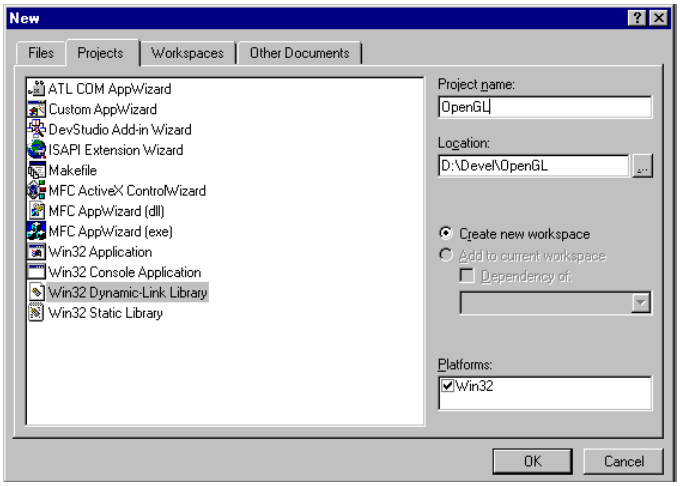
```
%{
#include <GL/glaux.h>
%}
%include "glaux.h"
```

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

Notes

Creating a Project

Extensions are compiled as DLLs



- Simply select a Win32 DLL when creating a new project.

Notes

On Unix, a shared library will be created using a Makefile similar to the following (will vary on every machine)

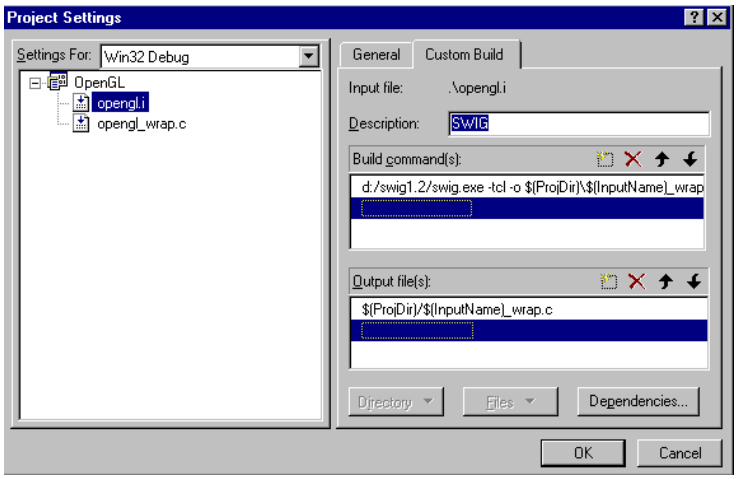
```
# Makefile for OpenGL (linux)
INTERFACE     = opengl.i
WRAPFILE      = $(INTERFACE:.i=_wrap.c)
WRAPOBJ       = $(INTERFACE:.i=_wrap.o)
TARGET        = opengl.so  # Use this kind of target for dynamic loading
CC            = gcc
CFLAGS        =
INCLUDE       = -I/usr/local/src/Mesa-2.5
LIBS          = -L/usr/local/src/Mesa-2.5/lib -lMesaaux -lMesatk -lMesaGLU -lMesaGL -lXext
OBJS          =

# SWIG Options
SWIG          = swig1.2
SWIGOPT       = -tcl

# Shared libraries
CCSHARED      = -fpic
BUILD         = gcc -shared

# Tcl installation (where is Tcl/Tk located)

TCL_INCLUDE   = -I/usr/include
TCL_LIB       = -L/usr/local/lib

all: $(TARGET)
# Create opengl_wrap.o from opengl_wrap.c
$(WRAPOBJ) : $(WRAPFILE)
        $(CC) -c $(CCSHARED) $(CFLAGS) $(WRAPFILE) $(INCLUDE) $(TCL_INCLUDE)
# Create the opengl_wrap.c from an interface file
$(WRAPFILE) : $(INTERFACE)
        $(SWIG) $(SWIGOPT) -o $(WRAPFILE) $(SWIGLIB) $(INTERFACE)
# Create the shared library
$(TARGET): $(WRAPOBJ) $(OBJS)
        $(BUILD) $(WRAPOBJ) $(OBJS) $(LIBS) -o $(TARGET)
```

Setting up the Files

The module consists of the following two files

- opengl.i (SWIG input)
- opengl_wrap.c (SWIG output)

Add both files to the project and customize opengl.i as follows



Notes

To customize the opengl.i, simply select the “project->settings” menu in Visual C++.

The full SWIG command line might look like the following:

```
d:\swig1.2\swig.exe -tcl -o $(ProjDir)\$(InputName)_wrap.c
          -I”d:\Program Files\DevStudio\Vc\include\GL” $(InputPath)
```

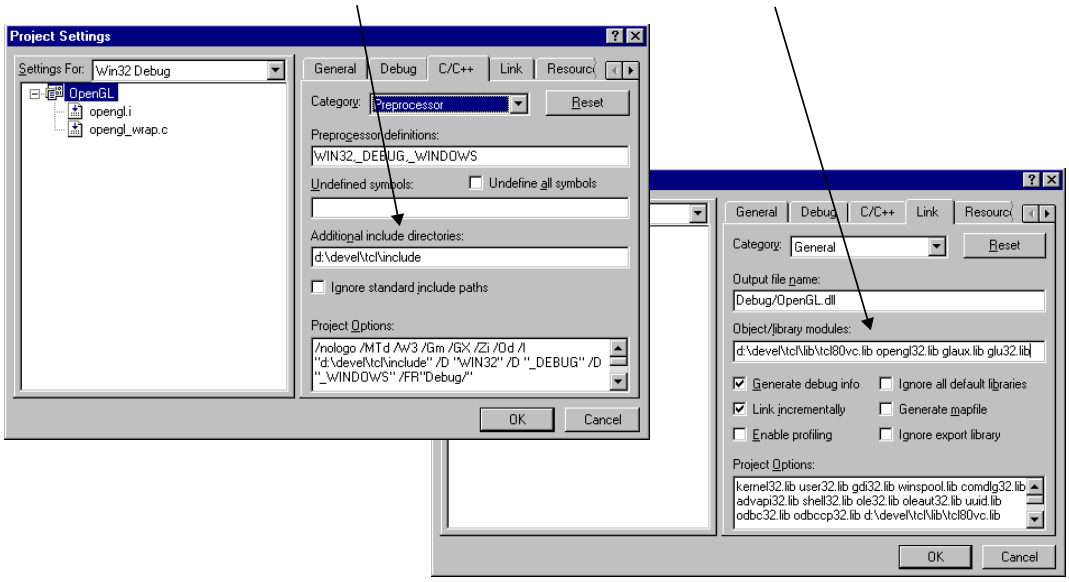
The output files should look like this :

```
$(ProjDir)/$(InputName)_wrap.c
```

Note, the file “opengl_wrap.c” should be added to the project even though it does not exist yet (Visual C++ will notice that the file doesn’t exist, but will ask you if its okay to proceed anyways).

Include Files and Libraries

Don’t forget the Tcl include directory and libraries



- Whew. Almost ready to give it a try.

Notes

First Attempt

Building our module now results in the following :

```
SWIG
Making wrappers for Tcl
d:\Program Files\DevStudio\VC\include\GL/gl.h : Line 1135. Syntax error in input.
d:\Program Files\DevStudio\VC\include\GL/gl.h : Line 1136. Syntax error in input.
d:\Program Files\DevStudio\VC\include\GL/gl.h : Line 1137. Variable WINGDIAPI multiply
defined (2nd definition ignored).
d:\Program Files\DevStudio\VC\include\GL/gl.h : Line 1137. Syntax error in input.
d:\Program Files\DevStudio\VC\include\GL/gl.h : Line 1138. Variable WINGDIAPI multiply
defined (2nd definition ignored).
d:\Program Files\DevStudio\VC\include\GL/gl.h : Line 1138. Syntax error in input.
d:\Program Files\DevStudio\VC\include\GL/gl.h : Line 1139. Variable WINGDIAPI multiply
defined (2nd definition ignored).
d:\Program Files\DevStudio\VC\include\GL/gl.h : Line 1139. Syntax error in input.
d:\Program Files\DevStudio\VC\include\GL/gl.h : Line 1140. Variable WINGDIAPI multiply
defined (2nd definition ignored).
d:\Program Files\DevStudio\VC\include\GL/gl.h : Line 1140. Syntax error in input.
d:\Program Files\DevStudio\VC\include\GL/gl.h : Line 1141. Variable WINGDIAPI multiply
defined (2nd definition ignored).
d:\Program Files\DevStudio\VC\include\GL/gl.h : Line 1141. Syntax error in input.
...
Confused by earlier errors. Bailing out
```

Hmmm. This isn’t too encouraging.

Notes

Fixing Parsing Problems

Raw C header files are often problematic. For example:

```
WINGDIAPI void APIENTRY glAccum(GLenum op, GLfloat val);
```

- SWIG has no idea what to do with macros or extensions to ANSI C.

Can use the SWIG preprocessor to fix many of these problems

```
// OpenGL Interface
%module opengl
// Define macros as empty (not needed for SWIG)
#define WINGDIAPI
#define APIENTRY
#define CALLBACK

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


Notes

Second Attempt

Getting much closer, only 3 errors this time

```
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 APIENTRY gluQuadricCallback (
//    GLUquadric   *qobj,
//    GLenum        which,
//    void          (CALLBACK *fn)();
```


Notes

Function pointers are not correctly parsed by SWIG1.1. SWIG1.2 will eventually eliminate these problems.

Pointers to functions can sometimes be handled using typedef. For example, the declaration

```
void foo(void (*pfcn)(int,int));
```

can be rewritten as

```
typedef void (*PFI)(int,int);
void foo(PFI pfcn);
```

Pointers to function may be awkward or difficult to use from a Tcl interface. While pointers to C functions can be passed around in Tcl, it is not possible to implement callback functions in Tcl or use Tcl procedures in place of C functions (well, not without a little work).

Third Attempt

The module now compiles

- Only had to make a few minor changes to headers

But the module still doesn't seem to work quite right...

```
% load ./opengl.dll
% auxInitDisplayMode [expr {$AUX_SINGLE | $AUX_RGBA |
          $AUX_DEPTH}]
% auxInitPosition 0 0 500 500
% auxInitWindow "Lit-Torus"
ambiguous command name "AuxInitWindow": auxInitWindowA
    auxInitWindowW
```

Header files and libraries sometimes play tricks

```
#ifdef UNICODE
#define auxInitWindow auxInitWindowW
#else
#define auxInitWindow auxInitWindowA
#end
GLenum APIENTRY auxInitWindowA(LPCSTR);
GLenum APIENTRY auxInitWindowW(LPCWSTR);
```


Notes

Wrapping a raw header file might not result in a usable Tcl extension module.

SWIG does not create wrappers for C macros as shown above.

Wrapping Macros

To wrap macros, simply supply a C prototype (with type information)

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

// Clear the macro definition
#undef auxInitWindow
// Give SWIG a C prototype for the macro
GLenum auxInitWindow(char *title);
```

May need to look at interface files to identify other macros

Notes

Type Problems

SWIG only really understands a few basic datatypes

- `int`, `long`, `short`, `float`, `double`, `char`, `void`
- Everything else is assumed to be a pointer

Missing typedef's are sometimes a problem

```
GLenum APIENTRY auxInitWindowA(LPCSTR);
```

- SWIG assumes LPCSTR is a complex object and creates a wrapper like this

```
GLenum wrap_auxInitWindowA(LPCSTR *a) {
       auxInitWindowA(*a);
}
```

- However, buried deep in Windows header files we find that LPCSTR is really a string

```
#define CONST  const
typedef char CHAR;
typedef CONST CHAR *LPCSTR;
```

To fix, put a typedef in the interface file

```
typedef const char *LPCSTR;
```


Notes

Helper Functions

Some functions may be difficult to use from Tcl

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

- 'params' is supposed to be an array.
- How do we manufacture these arrays in Tcl 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 *);
```

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

Notes

Tcl OpenGL Example

```
load ./opengl.dll
# Open up a display window
auxInitDisplayMode [expr {$AUX_SINGLE | $AUX_RGBA |
$AUX_DEPTH }]
auxInitPosition 0 0 500 500
auxInitWindow "Lit-Torus"

# Set up the material properties
set mat_specular [newfv4 1.0 1.0 1.0 1.0]
set mat_shininess [newfv4 50.0 0 0 0]
set light_position [newfv4 1.0 1.0 1.0 0.0]

glMaterialfv $GL_FRONT $GL_SPECULAR $mat_specular
glMaterialfv $GL_FRONT $GL_SHININESS $mat_shininess
glLightfv $GL_LIGHT0 $GL_POSITION $light_position
glEnable $GL_LIGHTING
glEnable $GL_LIGHT0
glDepthFunc $GL_LEQUAL
glEnable $GL_DEPTH_TEST

# Set up view
glClearColor 0 0 0 0
glColor3f 1.0 1.0 1.0
glMatrixMode $GL_PROJECTION
glLoadIdentity
glOrtho -1 1 -1 1 -1 1
glMatrixMode $GL_MODELVIEW
glLoadIdentity

glClear $GL_COLOR_BUFFER_BIT
glClear $GL_DEPTH_BUFFER_BIT
auxSolidTorus 0.10 0.50
```

Notes

Putting it All Together

Interface building is often an iterative process

- Start with header files
- Fix parsing problems and make slight edits (as necessary).
- Refine the interface to make it more usable.

Can be a very rapid process. For OpenGL:

- Had to define 3 macros.
- Edit out some function pointers.
- Supply a few typedefs.
- Write a few helper functions.

Some things to think about

- Raw headers might create an unusable interface.
- It is rarely necessary to wrap everything.
- Nothing was Tcl specific!

```
%module opengl

// Define problematic macros
#define APIENTRY
#define WINGDAPI
#define CALLBACK

// Provide a typedef
typedef const char *LPCSTR;

%include gl.i
%include glu.i
%include glaux.i
%include help.i

// Create a macro wrapper
#undef auxInitWindow
GLenum auxInitWindow(char *title);
...
```


Notes

In this example, a Tcl interface to OpenGL was built, but no Tcl specific code was written. As a result, it is easy to retarget our interface for other languages. For example :

```
swig -python opengl.i                # Build a Python interface to OpenGL
swig -perl5  opengl.i                # Build a Perl5 interface to OpenGl
```

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 Tcl?
- How do you access the internals of C/C++ objects in Tcl?
- How do you invoke C++ member functions from Tcl?

Concerns

- Don’t want to have to write a full C++ compiler to make it work (a nightmare).
- Don’t want to turn Tcl into C++.
- Don’t want to turn C++ into Tcl.
- Keep it simple.

Notes

Creating and Destroying Objects

Objects can be created and destroyed by writing helper 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

Using these functions in Tcl is straightforward:

```
% set v [new_Vector 1 -3 10]
% set w [new_Vector 0 -2.5 3]
% puts $v
_1100ef00_Vector_p
% puts $w
_1100ef20_Vector_p
% puts [dot_product $v $w]
37.5
% set a [cross_product $v $w]
% puts $a
_1100ef80_Vector_p
% delete_Vector $v
% delete_Vector $w
% delete_Vector $a
```

SWIG requires all objects to be explicitly created and destroyed. 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 internally. It would be a bad idea for Tcl to destroy an object that was still being used inside a C program. Unfortunately, there is no way for Tcl to know this.
- A C/C++ program may be performing its own management (reference counting, smart pointers, etc...). Tcl wouldn't know about this.

Accessing the Internals of an Object

This is 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

Invoking 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 are just creating 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;
};
```

SWIG

```
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

The creation of accessor functions is so straightforward, it makes sense for SWIG to automate the process.

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 Tcl?)
- 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

It is important to remember that SWIG only turns object definitions into accessor functions. This transformation can be easily performed with incomplete or partial information about the real C/C++ object. Again, SWIG is avoiding the problem of object data representation and using a scheme that relies upon references.

Compare with CORBA, COM, and other systems.

C++ Inheritance and Pointers

SWIG is aware of 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();
};
```

```
% set c [new_Circle 7]
% set s [new_Square 10]
% puts [Square_area $s]
100.0
% puts [Shape_area $s]
100.0
% puts [Shape_area $c]
153.938040046
puts [Square_area $c]
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.
- Multiple inheritance is also supported.

Notes

The Object Interface

SWIG can also create object-like Tcl interfaces

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

```
% Stack s                    ; # Create 's'
% s push Dave
% s push John
% s push Michelle
% s pop
Michelle
% puts [s cget -depth]
2
% puts [s cget -this]
_1008fe8_Stack_p
% rename s ""                ;# Delete
```

- Class is manipulated like a widget
- Data members accessed and modified using cget and configure
- Interface is somewhat similar to [incr Tcl].
- Object interface is built using low-level accessor functions.
- Many more details in the SWIG manual.

Note : SWIG is not an object-oriented extension to Tcl.

- For instance, you can't inherit from a SWIG object.

Notes

Extending Structures and Classes

Object extension : A cool trick for building Tcl interfaces

- You can provide additional “methods” for use only in Tcl
- Useful for debugging

```
%module example
struct Vector {
     double x,y,z;
};

// Attach some new methods to Vector
%addmethods Vector {
    Vector(double x, double y, double z) {
         Vector *v = new Vector;
         v->x = x;
         v->y = y;
         v->z = z;
         return v;
    }
    void output() {
         printf(“[%g, %g, %g]\n”,
                 self->x,self->y,self->z);
    }
};
```

```
# Create a vector
Vector v 1.0 2.0 -3.5

# Output its value
v output
```


Notes

The object extension mechanism works with both C and C++. Furthermore, it does not rely upon any C++ magic nor does it affect the underlying objects in any way.

Turning C Structures into Tcl Objects

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);
...
```

```
%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);
     }
};
```

SWIG

Tcl

```
# Create an image
Image img 500 500

# Manipulate it
img clear $BLACK
img plot 200 200 $WHITE
...
```

Can radically alter the Tcl interface to existing C programs.

Notes

Objects and Pointers

The object-interface is built using pointers and accessor functions

- However, objects and pointers are generally not interchangable.
- An “object” is simply a name---pointer value is hidden away in clientData.

Extracting the pointer value from an object

```
set ptr [obj cget -this]
```

- Extracts the pointer value from an object named ‘`obj`’

Converting a pointer value into an object

```
Object obj -this $ptr
```

- Creates an object named ‘`obj`’ from the pointer value `$ptr`.

More details in the SWIG manual

Notes

The SWIG Library

The SWIG Library

SWIG is packaged with a standard “library”

- Think of it as the SWIG equivalent of the C library.
- It’s an essential part of using SWIG.

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, Tcl 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

Rebuilding tclsh and wish

An alternative to dynamic loading

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

%include tclsh.i
// Use %include wish.i for wish
```

```
unix > swig -tcl example.i
unix > gcc example.c example_wrap.c \
     -I/usr/local/include -ltcl -lm \
      -o mytclsh
```

- `tclsh.i` and `wish.i` are SWIG library files for rebuilding the Tcl interpreter.
- Just run SWIG, compile, and link.
- No need to write `Tcl_AppInit()` or `main()`.

Can also specify libraries on the SWIG command line

```
swig -tcl -lwish.i example.i
```

- Makes it easier to write makefiles that use either dynamic or static linking.

Notes

The SWIG Pointer Library

%include pointer.i

- Provides high level creation, manipulation, and destruction of common C 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);
```

```
set a [ptrcreate double 3.5]
set b [ptrcreate double 7.0]
set c [ptrcreate double 0.0]
add $a $b $c
puts [ptrvalue $c]
10.5
ptrset $a -2.0
puts [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).

Typemap Library

The typemap library customizes SWIG (described shortly)

- Can be used to handle input and output arguments.

```
%module example
%include typemaps.i

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

Tcl

```
% load ./example.so
% set r [add 3.0 4.5]
% puts $r
7.5
%
```

- The behavior of "`double *INPUT`" and "`double *OUTPUT`" have been modified.
- Use of the C function has been completely changed (no longer requires pointers).

Notes

Support Files

Need a Tcl Makefile in a hurry?

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

- Copies a preconfigured Tcl 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 Tcl/Tk Makefile
#
# This file can be used to build various Tcl 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 ...
```

Advanced SWIG Features

Exception Handling

The %except directive

- Allows you to define an application specific exception handler.
- Can catch C errors or C++ exceptions.
- Fully configurable (you can define exception handlers as you wish).

```
%except(tcl) {
     try {
         $function   /* This gets replaced by the real function call */
     }
     catch(RangeError) {
          interp->result = "Array index out-of-bounds.";
          return TCL_ERROR;
     }
}
```

- Exception handling code gets inserted into all of the wrapper functions.
- `$function` token is replaced by the real C/C++ function call.

Note:

- Exception handling is different in Tcl 7.x and Tcl 8.x

Notes

SWIG includes a library of exception handlers that are implemented in a portable manner. To use the library, you would simply write the following :

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

In this case, the macro SWIG_exception() is translated into the appropriate Tcl code needed to indicate an error.

Typemaps

Typemaps allow you to change the processing of any datatype

- Handling of input/output values.
- Converting Tcl objects into C/C++ equivalents.
- Telling SWIG to use new Tcl types.

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 a hot topic of discussion on the SWIG mailing list

Caveats

- Requires knowledge of Tcl's C API to use effectively.
- It's possible to break SWIG in bizarre ways.
- Impossible to cover in full detail here.

Notes

Typemap Example

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,"")
```

- Can define rules for specific (datatype,name) pairs
- For example

```
%typemap(tcl,in) int a {
  if (Tcl_GetInt(interp,$source,$target)==TCL_ERROR)
     return TCL_ERROR;
  printf("a = %d\n", $target);
}
```

- This C code gets inserted into the wrapper functions created by SWIG.
- `$source` and `$target` are tokens that get replaced with C variables.
- The "in" typemap is used to convert arguments from Tcl to C.

Notes

How Typemaps Work

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

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

typemap →

```
static int _wrap_spam(ClientData clientData, Tcl_Interp *interp,
                      int argc, char *argv[]) {
    double  _result;
    int  _arg0;
    int  _arg1;

    clientData = clientData; argv = argv;
    if ((argc < 3) || (argc > 3)) {
        Tcl_SetResult(interp, "Wrong # args. spam a { int  } ",TCL_STATIC);
        return TCL_ERROR;
    }
    {
      if (Tcl_GetInt(interp,argv[1],_arg0)==TCL_ERROR)
          return TCL_ERROR;
      printf("a = %d\n", _arg0);
    }
    _arg1 = (int ) atol(argv[2]);
    _result = (double )spam(_arg0,_arg1);
    Tcl_PrintDouble(interp,(double) _result, interp->result);
    return TCL_OK;
}
```


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.

The bottom line

- Typemaps are very powerful and very useful.
- Can customize SWIG in a variety of ways.
- But you have to know what you’re doing.

Notes

Limitations

SWIG Limitations

Not a full C/C++ parser

- C++ function overloading.
- C++ operator overloading.
- Namespaces.
- Templates.
- Variable length arguments.
- Pointers to functions and pointers to arrays.
- Pointers to member functions.
- Problems with const.
- Can sometimes be difficult to track down parsing and code generation bugs.
- May be very hard to use with very complex header files.

May experience trouble with very large packages

- C++ systems with hundreds of header files.
- May generate an excessive amount of code (tens to hundreds of thousands of lines)

Notes

SWIG Limitations (cont...)

Integration with Tcl

- SWIG tries to keep a strict separation between C/C++ and Tcl.
- Appropriate for many applications.
- However, you might want more flexibility or control.
- SWIG typemaps can sometimes be used.
- Other Tcl extension building tools may work better--especially with object-oriented systems.
- SWIG can be used with other extension tools if you know what you are doing.

Notes

Application Troubles

Not all C/C++ applications work well in a scripted environment

- May crash.
- May operate in an unreliable manner.
- Very complex C++ systems may be very problematic (compilation, linking, etc...)
- An API may be poorly suited to scripting (i.e., difficult to use).
- Memory management woes.

Applications can also do bad things

- Example :

```
char *strcpy(char *s, const char *ct);
```

- Using this function from Tcl (as is) will likely corrupt the Tcl interpreter and crash. (Left as an exercise to the reader to figure out why).

Namespace clashes

- C/C++ functions may clash with Tcl commands.
- May clash with the C/C++ implementation of Tcl or other extensions.

Notes

An excellent overview of building C++ Tcl extensions is available at

```
http://zeus.informatik.uni-frankfurt.de/~fp/Tcl
```

Things to Keep in Mind

Extension building tools don’t necessarily result in a good interface

- Trying to wrap a raw header file is not guaranteed to work.
- Wrapping every function in a package is rarely necessary.
- A little planning and design go a long way.

SWIG may be inappropriate for very large projects

- Designed to be relatively informal and easy to use.
- Informality may be the exact opposite of what you want on a large project.

Shop around

- We are fortunate that Tcl is so easy to extend and that there are many tools.

There is no silver bullet

Notes

Summary

An Approach That Works

Extension building tools are being used in a variety of projects

- Simplify construction of large scripting interfaces.
- Improve productivity.
- Result in better applications.

A sampling of SWIG applications (from a user survey, 2/98)

Animation	Financial	Palm Pilot
Astrophysics	Fortran	Parallel computing
Automotive R&D	Games	Partial differential equation solvers
CAD Tools	Groupware	Polarization microscopy
CASE Tools	Hardware control/monitoring	Protein sequence analysis
COM	Image processing	PythonWin
CORBA	Integrated Development Environ.	Raytracing
Chemical information systems	Java	Realtime automation
Climate modeling	Lotus Notes	Robotics
Computational chemistry	Materials Modeling	Software testing
Computational steering	Medical Imaging	Spectrographic analysis
Database	Meteorological imaging	Speech recognition
Defibrillation modeling	Microprocessor design	Testing of telecom software
Document management	Military visualization	Virtual reality
Drawing	Molecular dynamics	Vision
Economics	Natural language processing	Visual simulation
Education	Network management	Weather forecasting
Electronic Design Automation	Neural nets	X-ray astrophysics analysis
Electronic Commerce	Oil exploration	

Notes

SWIG Resources

Web-page

`http://www.swig.org`

Includes links to other extension building tools and general resources.

FTP-server

`ftp://ftp.swig.org`

Mailing list

`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 Perl and Python so don't let the size scare you).

A variety of papers, tutorials, and other resources are also available.