SWIG and Tcl 10

This chapter discusses SWIG's support of Tcl. SWIG supports Tcl versions 7.3 and newer, including Tcl 8.0. Tk 3.6 and newer can also be used. However, for the best results you should consider using Tcl 7.5/Tk4.1 or later.

Preliminaries

You will need to install Tcl/Tk on your system if you haven't done so already. If you are using Tcl 7.5 or newer, you should also determine if your system supports dynamic loading and shared libraries. SWIG will work with or without this, but the compilation process varies.

Running SWIG

To build a Tcl module, run swig using the `-tcl` option as follows :

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

This will produce 2 files. The first file, `example_wrap.c`, contains all of the C code needed to build your Tcl module. The second file contains supporting documentation and may be named `example_wrap.doc`, `example_wrap.html`, `example_wrap.tex`, etc... To build a Tcl extension you will need to compile the `example_wrap.c` file and link it with the rest of your program (and possibly Tcl itself).

Additional SWIG options

The following options are also available with the Tcl module :

```
-tcl8         Produce Tcl8.0 native wrappers (use in place of -tcl).
-module       Set the module name.
-namespace    Use [incr Tcl] namespaces.
-prefix pkg   Set a package prefix of 'pkg'. This prefix will be
              attached to each function.
-htcl tcl.h   Set name of Tcl header file.
-htk tk.h     Set name of Tk header file.
-plugin       Generate additional code for the netscape plugin.
-noobject     Omit object oriented extensions (compatibility with SWIG 1.0)
```

Many of these options will be described later.

Getting the right header files and libraries

In order to compile Tcl/Tk extensions, you will need to locate the "`tcl.h`" and "`tk.h`" header files. These are usually located in `/usr/local/include`. You will also need to locate the Tcl/

Tk libraries `libtcl.a` and `libtk.a`. These are usually located in `/usr/local/lib`.
When locating the right header and libraries files, double check to make sure the files are the correct version and form a matching pair. SWIG works with the following Tcl/Tk releases.

```
Tcl 7.3, Tk 3.6
Tcl 7.4, Tk 4.0
Tcl 7.5, Tk 4.1
Tcl 7.6, Tk 4.2
Tcl 8.0a2, Tk 8.0a2
```

Do not mix versions. Although the code might compile if you do, it will usually core dump mysteriously. By default, SWIG looks for the header files "`tcl.h`" and "`tk.h`", but your installed version of Tcl/Tk may use slightly different names such as "`tcl7.5.h`" and "`tk4.1.h`". If you need to use different header files, you can use the `-htcl` and `-htk` options as in :

```
swig -tcl -htcl tcl7.5.h -htk tk4.1.h example.i
```

If you are installing Tcl/Tk yourself, it is often easier to set up a symbolic links between `tcl.h` and the header files for the latest installed version. You might also be able to make symbolic links to the correct files in your working directory.

Compiling a dynamic module (Unix)

To compile a dynamically loadable module, you will need to compile your SWIG extension into a shared library. This usually looks something like the following (shown for Linux).

```
unix > swig -tcl example.i
unix > gcc -fpic example_wrap.c example.c -I/usr/local/include
unix > gcc -shared example.o example_wrap.o -o example.so       # Linux
```

Unfortunately, the process of building of building shared libraries varies on every single machine. SWIG will try to guess when you run configure, but it isn't always successful. It's always a good idea to read the man pages on the compiler/linker to find out more information.

Using a dynamic module

To use a dynamic module, you will need to load it using the Tcl load command as follows :

```
load ./example.so example
```

The first argument is the name of the shared library while the second argument is the name of the module (the same as what you specified with the `%module` directive). As alternative, you can turn your module into a Tcl package. See the section on configuration management at the end of this chapter for details.

Static linking

If your machine does not support dynamic loading or you've tried to use it without success, you can build new versions of `tclsh` (the Tcl shell) or `wish` (Tcl/Tk shell) with your extensions added. To do this, use SWIG's `%include` directive as follows :

```
%module mymodule
... declarations ...
```

```
%include tclsh.i      // Support code for rebuilding tclsh
```

To rebuild tclsh, you will need to compile as follows :

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

Alternatively, you can use SWIG's `-l` option to add the tclsh.i library file without modifying the interface file. For example :

```
unix > swig -tcl -ltclsh.i example.i
unix > gcc example_wrap.c example.c -I/usr/local/include -L/usr/local/lib -ltcl -ldl \
       -lm -o my_tclsh
```

The `-ldl` option will be required if your Tcl/Tk supports dynamic loading. On some machines (most notably Solaris),it will also be necessary to add `-lsocket -lnsl` to the compile line. This will produce a new version of `tclsh` that is identical to the old one, but with your extensions added.

If you are using Tk, you will want to rebuild the `wish` executable instead. This can be done as follows :

```
%module mymodule
... declarations ...

%include wish.i       // Support code for rebuilding wish
```

The compilation process is similar as before, but now looks like this :

```
unix > swig -tcl example.i
unix > gcc example_wrap.c example.c -I/usr/local/include -L/usr/local/lib -ltk -ltcl \
       -lX11 -ldl -lm -o my_wish
```

In this case you will end up with a new version of the `wish` executable with your extensions added. Make sure you include `-ltk`, `-ltcl`, and `-lX11` in the order shown.

Compilation problems

Tcl is one of the easiest languages to compile extensions for. The Tcl header files should work without problems under C and C++. Perhaps the only tricky task is that of compiling dynamically loadable modules for C++. If your C++ code has static constructors, it is unlikely to work at all. In this case, you will need to build new versions of the `tclsh` or `wish` executables instead. You may also need to link against the `libgcc.a`, `libg++.a`, and `libstdc++.a` libraries (assuming g++).

Setting a package prefix

To avoid namespace problems, you can instruct SWIG to append a package prefix to all of your functions and variables. This is done using the -prefix option as follows :

```
swig -tcl -prefix Foo example.i
```

If you have a function "`bar`" in the SWIG file, the prefix option will append the prefix to the

name when creating a command and call it "`Foo_bar`".

Using [incr Tcl] namespaces

Alternatively, you can have SWIG install your module into an [incr Tcl] namespace by specifying the `-namespace` option :

```
swig -tcl -namespace example.i
```

By default, the name of the namespace will be the same as the module name, but you can override it using the `-prefix` option.

When the `-namespace` option is used, the resulting wrapper code can be compiled under both Tcl and [incr Tcl]. When compiling under Tcl, the namespace will turn into a package prefix such as in `Foo_bar`. When running under [incr Tcl], it will be something like `Foo::bar`.

Building Tcl/Tk Extensions under Windows 95/NT

Building a SWIG extension to Tcl/Tk under Windows 95/NT is roughly similar to the process used with Unix. Normally, you will want to produce a DLL that can be loaded into tclsh or wish. This section covers the process of using SWIG with Microsoft Visual C++ 4.x although the procedure may be similar with other compilers.

Running SWIG from Developer Studio

If you are developing your application within Microsoft developer studio, SWIG can be invoked as a custom build option. The process roughly follows these steps :

- Open up a new workspace and use the AppWizard to select a DLL project.
- Add both the SWIG interface file (the .i file), any supporting C files, and the name of the wrapper file that will be created by SWIG (ie. `example_wrap.c`). Note : If using C++, choose a different suffix for the wrapper file such as `example_wrap.cxx`. Don't worry if the wrapper file doesn't exist yet--Developer studio will keep a reference to it around.
- Select the SWIG interface file and go to the settings menu. Under settings, select the "Custom Build" option.
- Enter "SWIG" in the description field.
- Enter "`swig -tcl -o $(ProjDir)\$(InputName)_wrap.c $(InputPath)`" in the "Build command(s) field"
- Enter "`$(ProjDir)\$(InputName)_wrap.c`" in the "Output files(s) field".
- Next, select the settings for the entire project and go to "C++:Preprocessor". Add the include directories for your Tcl installation under "Additional include directories".
- Finally, select the settings for the entire project and go to "Link Options". Add the Tcl library file to your link libraries. For example "`tcl80.lib`". Also, set the name of the output file to match the name of your Tcl module (ie. example.dll).
- Build your project.

Now, assuming all went well, SWIG will be automatically invoked when you build your project. Any changes made to the interface file will result in SWIG being automatically invoked to produce a new version of the wrapper file. To run your new Tcl extension, simply run `tclsh` or `wish` and use the `load` command. For example :

```
MSDOS > tclsh80
% load example.dll
% fact 4
24
%
```

Using NMAKE

Alternatively, SWIG extensions can be built by writing a Makefile for NMAKE. To do this, make sure the environment variables for MSVC++ are available and the MSVC++ tools are in your path. Now, just write a short Makefile like this :

```
# Makefile for building various SWIG generated extensions

SRCS          = example.c
IFILE         = example
INTERFACE     = $(IFILE).i
WRAPFILE      = $(IFILE)_wrap.c

# Location of the Visual C++ tools (32 bit assumed)

TOOLS         = c:\msdev
TARGET        = example.dll
CC            = $(TOOLS)\bin\cl.exe
LINK          = $(TOOLS)\bin\link.exe
INCLUDE32     = -I$(TOOLS)\include
MACHINE       = IX86

# C Library needed to build a DLL

DLLIBC        = msvcrt.lib oldnames.lib

# Windows libraries that are apparently needed
WINLIB        = kernel32.lib advapi32.lib user32.lib gdi32.lib comdlg32.lib
winspool.lib

# Libraries common to all DLLs
LIBS          = $(DLLIBC) $(WINLIB)

# Linker options
LOPT      = -debug:full -debugtype:cv /NODEFAULTLIB /RELEASE /NOLOGO /
MACHINE:$(MACHINE) -entry:_DllMainCRTStartup@12 -dll

# C compiler flags

CFLAGS    = /Z7 /Od /c /nologo
TCL_INCLUDES  = -Id:\tcl8.0a2\generic -Id:\tcl8.0a2\win
TCLLIB        = d:\tcl8.0a2\win\tcl80.lib

tcl::
       ..\..\swig -tcl -o $(WRAPFILE) $(INTERFACE)
       $(CC) $(CFLAGS) $(TCL_INCLUDES) $(SRCS) $(WRAPFILE)
       set LIB=$(TOOLS)\lib
       $(LINK) $(LOPT) -out:example.dll $(LIBS) $(TCLLIB) example.obj example_wrap.obj
```

To build the extension, run NMAKE (you may need to run vcvars32 first). This is a pretty minimal Makefile, but hopefully its enough to get you started. With a little practice, you'll be making lots of Tcl extensions.

Basic Tcl Interface

Functions

C functions are turned into new Tcl commands with the same usage as the C function. Default/optional arguments are also allowed. An interface file like this :

```
%module example
int foo(int a);
double bar (double, double b = 3.0);
...
```

Will be used in Tcl like this :

```
set a [foo 2]
set b [bar 3.5 -1.5]
set b [bar 3.5]             # Note : default argument is used
```

There isn't much more to say...this is pretty straightforward.

Global variables

For global variables, things are a little more complicated. For the following C datatypes, SWIG will use Tcl's variable linking mechanism to provide direct access :

```
int, unsigned int,
double,
char *,
```

When used in an interface file and script, it will operate as follows :

```
// example.i
%module example
...
double My_variable;
...

# Tcl script
puts $My_variable           # Output value of C global variable
set My_variable 5.5         # Change the value
```

For all other C datatypes, SWIG will generate a pair of set/get functions. For example :

```
// example.i
short My_short;
```

will be accessed in Tcl as follows :

```
puts [My_short_get]               # Get value of global variable
My_short_set 5.5                  # Set value of global variable
```

While not the most elegant solution, this is the only solution for now. Tcl's normal variable linking mechanism operates directly on a variables and would not work correctly on datatypes other than the 3 basic datatypes supported by Tcl (`int`, `double`, and `char *`).

Constants

Constants are created as read-only variables. For odd datatypes (not supported by the variable linking mechanism), SWIG generates a string representation of the constant and use it instead (you shouldn't notice this however since everything is already a string in Tcl). It is never necessary to use a special "get" function with constants. Unlike Tcl variables, constants can contain pointers, structure addresses, function pointers, etc...

Pointers

Pointers to C/C++ objects are represented as character strings such as the following :

```
_100f8e2_Vector_p
```

A NULL pointer is represented by the string "NULL". NULL pointers can also be explicitly created as follows :

```
_0_Vector_p
```

In Tcl 8.0, pointers are represented using a new type of Tcl object, but the string representation is the same (and is interchangable). As a general, direct manipulation of pointer values is discouraged.

Structures

SWIG generates a basic low-level interface to C structures. For example :

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

gets mapped into the following collection of C functions :

```
double Vector_x_get(Vector *obj)
double Vector_x_set(Vector *obj, double x)
double Vector_y_get(Vector *obj)
double Vector_y_set(Vector *obj, double y)
double Vector_z_get(Vector *obj)
double Vector_z_set(Vector *obj, double z)
```

These functions are then used in the resulting Tcl interface. For example :

```
# v is a Vector that got created somehow
% Vector_x_get $v
3.5
% Vector_x_set $v 7.8              # Change x component
```

Similar access is provided for unions and the data members of C++ classes.

C++ Classes

C++ classes are handled by building a set of low level accessor functions. Consider the following class :

```
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);
};
```

When wrapped by SWIG, the following functions are created :

```
List    *new_List();
void     delete_List(List *l);
int      List_search(List *l, char *item);
void     List_insert(List *l, char *item);
void     List_remove(List *l, char *item);
char    *List_get(List *l, int n);
int      List_length_get(List *l);
int      List_length_set(List *l, int n);
void     List_print(List *l);
```

Within Tcl, we can use the functions as follows :

```
% set l [new_List]
% List_insert $l Ale
% List_insert $l Stout
% List_insert $l Lager
% List_print $l
Lager
Stout
Ale
% puts [List_length_get $l]
3
% puts $l
_1008560_List_p
%
```

C++ objects are really just pointers (which are represented as strings). Member functions and data are accessed by simply passing a pointer into a collection of accessor functions that take the pointer as the first argument.

While somewhat primitive, the low-level SWIG interface provides direct and flexible access to almost any C++ object. As it turns out, it is possible to do some rather amazing things with this interface as will be shown in some of the later examples. SWIG 1.1 also generates an object oriented interface that can be used in addition to the basic interface just described here.

The object oriented interface

With SWIG 1.1, a new object oriented interface to C structures and C++ classes is supported. This interface supplements the low-level SWIG interface already defined--in fact, both can be used simultaneously. To illustrate this interface, consider our previous `List` class :

```
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);
};
```

Using the object oriented interface requires no additional modifications or recompilation of the SWIG module (the functions are just used differently).

Creating new objects

The name of the class becomes a new command for creating an object. There are 5 methods for creating an object (`MyObject` is the name of the corresponding C++ class)

```
MyObject o                  # Creates a new object 'o'

MyObject o -this $objptr    # Turn a pointer to an existing C++ object into a
                            # Tcl object 'o'

MyObject -this $objptr      # Turn the pointer $objptr into a Tcl "object"

MyObject -args args         # Create a new object and pick a name for it. A handle
                            # will be returned and is the same as the pointer value.

MyObject                    # The same as MyObject -args, but for constructors that
                            # take no arguments.
```

Thus, for our List class, we can create new List objects as follows :

```
List l                      # Create a new list l

set listptr [new_List]      # Create a new List using low level interface
List l2 -this $listptr      # Turn it into a List object named 'l2'

set l3 [List]               # Create a new list. The name of the list is in $l3

List -this $listptr         # Turn $listptr into a Tcl object of the same name
```

Now assuming you're not completely confused at this point, the best way to think of this is that there are really two different ways of representing an object. One approach is to simply use the pointer value as the name of an object. For example :

```
_100e8f8_List_p
```

The second approach is to allow you to pick a name for an object such as “foo”. The different types of constructors are really just mechanism for using either approach.

Invoking member functions

Member functions are invoked using the name of the object followed by the method name and any arguments. For example :

```
% List l
% l insert “Bob”
% l insert “Mary”
% l search “Dave”
0
% ...
```

Or if you let SWIG generate the name of the object... (this is the pointer model)

```
% set l [List]
% $l insert “Bob”                  # Note $l contains the name of the object
% $l insert “Mary”
% $l search “Dave”
0
%
```

Deleting objects

Since objects are created by adding new Tcl commands, they can be deleted by simply renaming them. For example :

```
% rename l “”               # Destroy list object ‘l’
```

SWIG will automatically call the corresponding C/C++ destructor, with one caveat--SWIG will not destroy an object if you created it from an already existing pointer (if you called the constructor using the -this option). Since the pointer already existed when you created the Tcl object, Tcl doesn’t own the object so it would probably be a bad idea to destroy it.

Accessing member data

Member data of an object can be accessed using the `cget` method. The approach is quite similar to that used in [incr Tcl] and other Tcl extensions. For example :

```
% l cget -length            # Get the length of the list
13
```

The `cget` method currently only allows retrieval of one member at a time. Extracting multiple members will require repeated calls.

The member `-this` contains the pointer to the object that is compatible with other SWIG functions. Thus, the following call would be legal

```
% List l                          # Create a new list object
% l insert Mike
% List_print [l cget -this]       # Print it out using low-level function
```

Changing member data

To change the value of member data, the `configure` method can be used. For example :

```
% l configure -length 10     # Change length to 10 (probably not a good idea, but
                             # possible).
```

In a structure such as the following :

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

you can change the value of all or some of the members as follows :

```
% v configure -x 3.5 -y 2 -z -1.0
```

The order of attributes does not matter.

Relationship with pointers

The object oriented interface is mostly compatible with all of the functions that accept pointer values as arguments. Here are a couple of things to keep in mind :

- If you explicitly gave a name to an object, the pointer value can be retrieved using the '`cget -this`' method. The pointer value is what you should give to other SWIG generated functions if necessary.
- If you let SWIG generate the name of an object for you, then the name of the object is the same as the pointer value. This is the preferred approach.
- If you have a pointer value but it's not a Tcl object, you can turn it into one by calling the constructor with the '`-this`' option.

Here is a script that illustrates how these things work :

```
# Example 1 : Using a named object

List l                       # Create a new list
l insert Dave                # Call some methods
l insert Jane
l insert Pat
List_print [l cget -this]    # Call a static method (which requires the pointer value)

# Example 2: Let SWIG pick a name

set l [List]                 # Create a new list
$l insert Dave               # Call some methods
$l insert Jane
$l insert Pat
List_print $l                # Call static method (name of object is same as pointer)

# Example 3: Already existing object
set l [new_List]             # Create a raw object using low-level interface
List_insert $l Dave          # Call some methods (using low-level functions)
List -this $l                # Turn it into a Tcl object instead
$l insert Jane
```

```
$l insert Part
List_print $l               # Call static method (uses pointer value as before).
```

Performance concerns and disabling the object oriented interface

The object oriented interface is mainly provided for ease of programming at the expense of introducing more overhead and increased code size (C code that is). If you are concerned about these issues use the basic SWIG interface instead. It provides direct access and is much faster. As it turns out, it is possible to build an object oriented interface directly in Tcl as well--an example we'll return to a little later.

To disable the object oriented interface, run SWIG with the `-noobject` option. This will strip out all of the extra code and produce only the low-level interface.

About the examples

The next few sections cover Tcl examples of varying complexity. These are primarily designed to illustrate how SWIG can be used to integrate C/C++ and Tcl in a variety of ways. Some of the things that will be covered are :

- Controlling C programs with Tcl
- Building C data structures in Tcl.
- Use of C objects with Tk
- Wrapping a C library (OpenGL in this case)
- Accessing arrays and other common data structures
- Using Tcl to build new Tcl interfaces to C programs.
- Modifying SWIG's handling of datatypes.
- And a bunch of other cool stuff.

Binary trees in Tcl

In this example, we show Tcl can be used to manipulate binary trees implemented in C. This will involve accessing C data structures and functions.

C files

We will build trees using the following C header file :

```
/* tree.h */
typedef struct Node Node;
struct Node {
       char          *key;
       char          *value;
       Node          *left;
       Node          *right;
};

typedef struct Tree {
       Node          *head;        /*  Starting node */
       Node          *z;           /* Ending node (at leaves) */
} Tree;
```

```
extern Node *new_Node(char *key, char *value);
extern Tree *new_Tree();
```

The C functions to create new nodes and trees are as follows :

```
/* File : tree.c */
#include <string.h>
#include "tree.h"
Node *new_Node(char *key, char *value) {
       Node *n;
       n = (Node *) malloc(sizeof(Node));
       n->key = (char *) malloc(strlen(key)+1);
       n->value = (char *) malloc(strlen(value)+1);
       strcpy(n->key,key);
       strcpy(n->value,value);
       n->left = 0;
       n->right = 0;
       return n;
};
Tree *new_Tree() {
       Tree *t;
       t = (Tree *) malloc(sizeof(Tree));
       t->head = new_Node("","__head__");
       t->z = new_Node("__end__","__end__");
       t->head->right = t->z;
       t->z->left = t->z;
       t->z->right = t->z;
       return t;
}
```

Making a quick a dirty Tcl module

To make a quick Tcl module with these functions, we can do the following :

```
// file : tree.i
%module tree
%{
#include "tree.h"
%}
%include tree.h              // Just grab header file since it's fairly simple
```

To build the module, run SWIG as follows and compile the resulting output :

```
% swig -tcl -ltclsh.i tree.i
% gcc tree.c tree_wrap.c -I/usr/local/include -L/usr/local/lib -ltcl -lm -o my_tclsh
```

We can now try out the module interactively by just running the new '`my_tclsh`' executable.

```
unix > my_tclsh
% set t [new_Tree]                 # Create a new tree
_8053198_Tree_p
% set n [Tree_head_get $t]         # Get first node
_80531a8_Node_p
% puts [Node_value_get $n]         # Get its value
__head__
% Node -this $n
```

```
% $n cget -value                    # Altenative method for getting value
__head__
```

Building a C data structure in Tcl

Given our simple Tcl interface, it is easy to write Tcl functions for building up a C binary tree. For example :

```
# Insert an item into a tree
proc insert_tree {tree key value} {
    set tree_head [Tree_head_get $tree]
    set tree_z [Tree_z_get $tree]
    set p $tree_head
    set x [Node_right_get $tree_head]
    while {[Node_key_get $x] != "__end__"} {
        set p $x
        if {$key < [Node_key_get $x]} {
              set x [Node_left_get $x]
        } {
              set x [Node_right_get $x]
        }
    }
    set x [new_Node $key $value]
    if {$key < [Node_key_get $p]} {
        Node_left_set $p $x
    } {
        Node_right_set $p $x
    }
    Node_left_set $x $tree_z
    Node_right_set $x $tree_z
}

# Search tree and return all matches
proc search_tree {tree key} {
    set tree_head [Tree_head_get $tree]
    set tree_z [Tree_z_get $tree]
    set found {}
    set x [Node_right_get $tree_head]
    while {[Node_key_get $x] != "__end__"} {
        if {[Node_key_get $x] == $key} {
            lappend found [Node_value_get $x]
        }
        if {$key < [Node_key_get $x]} {
            set x [Node_left_get $x]
        } {
            set x [Node_right_get $x]
        }
    }
    return $found
}
```

While written in Tcl, these functions are building up a real C binary tree data structure that could be passed into other C function. For example, we could write a function that globs an entire directory and builds a tree structure as follows :

```
# Insert all of the files in pathname into a binary tree
```

```
proc build_dir_tree {tree pathname} {
    set error [catch {set filelist [glob -nocomplain $pathname/*]}]
    if {$error == 0} {
        foreach f $filelist {
            if {[file isdirectory $f] == 1} {
                insert_tree $tree [file tail $f] $f
                if {[file type $f] != "link"} {build_dir_tree $tree $f}
            } {
                insert_tree $tree [file tail $f] $f
            }
        }
    }
}
```

We can test out our function interactively as follows :

```
% source tree.tcl
% set t [new_Tree]          # Create a new tree
_80533c8_Tree_p
% build_dir_tree $t /home/beazley/SWIG/SWIG1.1
% search_tree $t tcl
/home/beazley/SWIG/SWIG1.1/Examples/tcl /home/beazley/SWIG/SWIG1.1/swig_lib/tcl
%
```

Implementing methods in C

While our Tcl methods may be fine for small problems, it may be faster to reimplement the insert and search methods in C :

```
void insert_tree(Tree *t, char *key, char *value) {
       Node *p;
       Node *x;

       p = t->head;
       x = t->head->right;
       while (x != t->z) {
              p = x;
              if (strcmp(key,x->key) < 0)
                     x = x->left;
              else
                     x = x->right;
       }
       x = new_Node(key,value);
       if (strcmp(key,p->key) < 0)
              p->left = x;
       else
              p->right = x;
       x->left = t->z;
       x->right = t->z;
}
```

To use this function in Tcl, simply put a declaration into the file `tree.h` or `tree.i`.

When reimplemented in C, the underlying Tcl script may not notice the difference. For example, our directory subroutine would not care if `insert_tree` had been written in Tcl or C. Of course, by writing this function C, we will get significantly better performance.

Building an object oriented C interface

So far our tree example has been using the basic SWIG interface. With a little work in the interface file, we can improve the interface a little bit.

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

%include tree.h
%{

  /* Function to count up Nodes */
  static  int count_nodes(Node *n, Node *end) {
    if (n == end) return 0;
    return 1+count_nodes(n->left,end)+count_nodes(n->right,end);
  }

%}

// Attach some new methods to the Tree structure

%addmethods Tree {
   void insert(char *key, char *value) {
        insert_tree(self,key,value);
   }
   char *search(char *key) {
        return search_tree(self,key);
   }
   char *findnext(char *key) {
     return find_next(self,key);
   }
   int count() {
     return count_nodes(self->head->right,self->z);
   }
   Tree();        // This is just another name for new_Tree
}
```

The `%addmethods` directive can be used to attach methods to existing structures and classes. In this case, we are attaching some methods to the `Tree` structure. Each of the methods are simply various C functions we have written for accessing trees. This type of interface file comes in particularly handy when using the Tcl object oriented interface. For example, we can rewrite our directory globber as follows :

```
proc build_dir_tree {tree pathname} {
    set error [catch {set filelist [glob -nocomplain $pathname/*]}]
    if {$error == 0} {
        foreach f $filelist {
            if {[file isdirectory $f] == 1} {
                $tree insert [file tail $f] $f      # Note new calling method
                if {[file type $f] != "link"} {build_dir_tree $tree $f}
            } {
                $tree insert [file tail $f] $f
            }
        }
    }
```

```
}
```

Now using it :

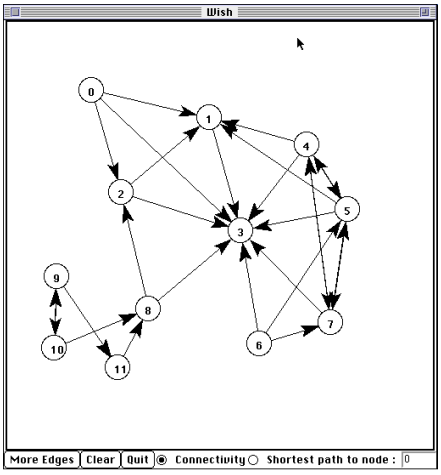
```
% source tree.tcl
% Tree t                    # Create a new Tree object
_8054610_Tree_p
% build_dir_tree t /home/beazley/SWIG/SWIG1.1
% t count
1770
% t search glaux.i
/home/beazley/SWIG/SWIG1.1/Examples/OpenGL/glaux.i
% t search typemaps
/home/beazley/SWIG/SWIG1.1/Examples/perl5/typemaps
% t findnext typemaps
/home/beazley/SWIG/SWIG1.1/Examples/python/typemaps
% t findnext typemaps
/home/beazley/SWIG/SWIG1.1/Examples/tcl/typemaps
% t findnext typemaps
None
%
```

With a little extra work, we've managed to turn an ordinary C structure into a class-like object in Tcl.

Building C/C++ data structures with Tk

Given the direct access to C/C++ objects provided by SWIG, it can be possible to use Tk to interactively build a variety of interesting data structures. To do this, it is usually useful to maintain a mapping between canvas items and an underlying C data structure. This is done using associative arrays to map C pointers to canvas items and canvas items back to pointers.

Suppose that we have a C program for manipulating directed graphs and that we wanted to provide a Tk interface for building graphs using a ball-and-stick model such as the following :



The SWIG interface file for this might look something like this :

```
%module graph
%{
```

```
#include "graph.h"
%}

/* Get a node's number */
int      GetNum(Node *n);                        /* Get a node's number         */
AdjList *GetAdj(Node *n);;                       /* Get a node's adjacency list */
AdjList *GetNext(AdjList *l);                    /* Get next node in adj. list  */
Node    *GetNode(AdjList *l);                    /* Get node from adj. list     */
Node    *new_node();                             /* Make a new node             */
void     AddLink(Node *v1, Node *v2);            /* Add an edge                 */
... etc ...
```

The interface file manipulates `Node` and `AdjList` structures. The precise implementation of these doesn't matter here--in fact SWIG doesn't even need it. Within a Tcl/Tk script however, we can keep track of these objects as follows :

```
# Make a new node and put it on the canvas
proc mkNode {x y} {
    global nodeX nodeY nodeP nodeMap nodeList edgeFirst edgeSecond
    set new [.c create oval [expr $x-15] [expr $y-15] \
                [expr $x+15] [expr $y+15] -outline black \
                -fill white -tags node]
    set newnode [new_node]                                  ;# Make a new C Node
    set nnum [GetNum $newnode]                              ;# Get our node's number
    set id [.c create text [expr $x-4] [expr $y+10] \
             -text $nnum -anchor sw -tags nodeid]
    set nodeX($new) $x                                      ;# Save coords of canvas item
    set nodeY($new) $y
    set nodeP($new) $newnode                                ;# Save C pointer
    set nodeMap($newnode) $new                              ;# Map pointer to Tk widget
    set edgeFirst($new) {}
    set edgeSecond($new) {}
    lappend nodeList $new                                   ;# Keep a list of all C
                                                            ;# Pointers we've made
}

# Make a new edge between two nodes and put an arrow on the canvas
proc mkEdge {first second new} {
    global nodeP edgeFirst edgeSecond
    set edge [mkArrow $first $second]                       ;# Make an arrow
    lappend edgeFirst($first) $edge                         ;# Save information
    lappend edgeSecond($second) $edge
    if {$new == 1} {
# Now add an edge within our C data structure
        AddLink $nodeP($first) $nodeP($second)              ;# Add link to C structure
    }
}
```

In these functions, the array `nodeP()` allows us to associate a particular canvas item with a C object. When manipulating the graph, this makes it easy to move from the Tcl world to C. A second array, `nodeMap()` allows us to go the other way--mapping C pointers to canvas items. A list `nodeList` keeps track of all of the nodes we've created just in case we want to examine all of the nodes. For example, suppose a C function added more edges to the graph. To reflect the new state of the graph, we would want to add more edges to the Tk canvas. This might be accomplished as follows :

```
# Look at the C data structure and update the canvas
proc mkEdges {} {
    global nodeX nodeY nodeP nodeMap nodeList edgeFirst edgeSecond
    unset edgeFirst
    unset edgeSecond
    .c delete arc              # Edges have been tagged with arc (not shown)

    foreach node $nodeList {   ;# clear old lists
       set edgeFirst($node) {}
       set edgeSecond($node) {}
    }
    foreach node $nodeList {                    ;# Go through all of the nodes
       set v $nodeP($node)                      ;# Get the node pointer
       set v1 [GetAdj $v]                       ;# Get its adjacency list
       while {$v1 != "NULL"} {
           set v2 [GetNode $v1]                 ;# Get node pointer
           set w $nodeMap($v2)                  ;# Get canvas item
           mkEdge $node $w 0                    ;# Make an edge between them
           set v1 [GetNext $v1]                 ;# Get next node
       }
    }
}
```

This function merely scans through all of the nodes and walks down the adjacency list of each one. The `nodeMap()` array maps C pointers onto the corresponding canvas items. We use this to construct edges on the canvas using the `mkEdge` function.

Accessing arrays

In some cases, C functions may involve arrays and other objects. In these instances, you may have to write helper functions to provide access. For example, suppose you have a C function like this :

```
// Add vector a+b -> c
void vector_add(double *a, double *b, double *c, int size);
```

SWIG is quite literal in its interpretation of `double *`--it is a pointer to a `double`. To provide access, a few helper functions can be written such as the following :

```
// SWIG helper functions for double arrays
%inline %{
double *new_double(int size) {
       return (double *) malloc(size*sizeof(double));
}
void delete_double(double *a) {
       free a;
}
double get_double(double *a, int index) {
       return a[index];
}
void set_double(double *a, int index, double val) {
       a[index] = val;
}
%}
```

Using our C functions might work like this :

```
# Tcl code to create some arrays and add them

set a [new_double 200]
set b [new_double 200]
set c [new_double 200]

# Fill a and b with some values
for {set i 0} {$i < 200} {incr i 1} {
       set_double $a $i 0.0
       set_double $b $i $i
}

# Add them and store result in c
vector_add $a $b $c 200
```

The functions `get_double` and `set_double` can be used to access individual elements of an array. To convert from Tcl lists to C arrays, one could write a few functions in Tcl such as the following :

```
# Tcl Procedure to turn a list into a C array
proc Tcl2Array {l} {
       set len [llength $l]
       set a [new_double $len]
       set i 0
       foreach item $l {
              set_double $a $i $item
              incr i 1
       }
       return $a
}

# Tcl Procedure to turn a C array into a Tcl List
proc Array2Tcl {a size} {
       set l {}
       for {set i 0} {$i < size} {incr i 1} {
              lappend $l [get_double $a $i]
       }
       return $l
}
```

While not optimal, one could use these to turn a Tcl list into a C representation. The C representation could be used repeatedly in a variety of C functions without having to repeatedly convert from strings (Of course, if the Tcl list changed one would want to update the C version). Likewise, it is relatively simple to go back from C into Tcl. This is not the only way to manage arrays--typemaps can be used as well. The SWIG library file '`array.i`' also contains a variety of pre-written helper functions for managing different kinds of arrays.

Building a simple OpenGL module

In this example, we consider building a SWIG module out of the OpenGL graphics library. The OpenGL library consists of several hundred functions for building complex 3D images. By wrapping this library, we will be able to play with it interactively from a Tcl interpreter.

Required files

To build an OpenGL module, you will need to have some variant of OpenGL installed on your machine. If unavailable, the Mesa graphics library is an OpenGL clone that runs on most machines that support X11. We will use the "GL/gl.h", "GL/glu.h", and the GL Auxilliary libraries.

Wrapping gl.h

The first step is to build an interface from the gl.h file. To do this, follow these steps :

- Copy the file `gl.h` to a file `gl.i` which will become the interface.
- Edit the `gl.i` file by taking out unneeded C preprocessor directives and any other clutter that you find.
- Put the following code at the beginning of the `gl.i` file

```
// gl.i : SWIG file for OpenGL
%module gl
%{
#include <GL/gl.h>
%}

... Rest of edited gl.h file here ...
```

A critical part of this first step is making sure you have the proper set of typedefs in the `gl.i` file. The first part of the file should include definitions such as the following :

```
typedef unsigned int GLenum;
typedef unsigned char GLboolean;
typedef unsigned int GLbitfield;
typedef signed char GLbyte;
typedef short GLshort;
typedef int GLint;
typedef int GLsizei;
typedef unsigned char GLubyte;
typedef unsigned short GLushort;
typedef unsigned int GLuint;
typedef float GLfloat;
typedef float GLclampf;
typedef double GLdouble;
typedef double GLclampd;
typedef void GLvoid;
```

Wrapping glu.h

Next we write a module for the glu library. The procedure is essentially identical to `gl.h`--that is, we'll copy the header file and edit it slightly. In this case, `glu.h` contains a few functions involving function pointers and arrays. These may generate errors or warnings. As a result, we can simply edit those declarations out. Fortunately, there aren't many declarations like this. If access is required later, the problem can often be fixed with typemaps and helper functions. The final `glu.i` file will look something like this :

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

```
%}

... rest of edited glu.h file here ...
```

Given these two files, we have a fairly complete OpenGL package. Unfortunately, we still don't have any mechanism for opening a GL window and creating images. To to this, we can wrap the OpenGL auxilliary library which contains functions to open windows and draw various kinds of objects (while not the most powerful approach, it is the simplest to implement and works on most machines).

Wrapping the aux library

Wrapping the aux library follows exactly the same procedure as before. You will create a file `glaux.i` such as the following :

```
// File :glaux.i
%module glaux
%{
#include "glaux.h"
%}

... Rest of edited glaux.h file ...
```

A few helper functions

Finally, to make our library a little easier to use, we need to have a few functions to handle arrays since quite a few OpenGL calls use them as arguments. Small 4 element arrays are particularly useful so we'll create a few helper functions in a file called `help.i`.

```
// help.i : OpenGL helper functions

%inline %{
GLfloat *newfv4(GLfloat a,GLfloat b,GLfloat c,GLfloat d) {
  GLfloat *f;
  f = (GLfloat *) malloc(4*sizeof(GLfloat));
  f[0] = a; f[1] = b; f[2] = c; f[3] = d;
  return f;
}
void setfv4(GLfloat *fv, GLfloat a, GLfloat b, GLfloat c, GLfloat d) {
  fv[0] = a; fv[1] = b; fv[2] = c; fv[3] = d;
}
%}
%name(delfv4) void free(void *);
```

An OpenGL package

Whew, we're almost done now. The last thing to do is to package up all of our interface files into a single file called `opengl.i`.

```
//
// opengl.i.    SWIG Interface file for OpenGL
%module opengl

%include gl.i         // The main GL functions
```

```
%include glu.i        // The GLU library
%include glaux.i      // The aux library
%include help.i       // Our helper functions
```

To build the module, we can simply run SWIG as follows :

```
unix > swig -tcl opengl.i           #  Build a dynamicly loaded extension
```

or

```
unix > swig -tcl -lwish.i opengl.i  # Build a statically linked wish executable
```

Compile the file `opengl_wrap.c` with your C compiler and link with Tcl, Tk, and OpenGL to create the final module.

Using the OpenGL module

The module is now used by writing a Tcl script such as the following :

```
load ./opengl.so
auxInitDisplayMode [expr {$AUX_SINGLE | $AUX_RGBA | $AUX_DEPTH}]
auxInitPosition 0 0 500 500
auxInitWindow "Lit-Sphere"

# set up 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

# Draw it
glClear $GL_COLOR_BUFFER_BIT
glClear $GL_DEPTH_BUFFER_BIT
auxSolidSphere 0.5

# Clean up
delfv4 $mat_specular
delfv4 $mat_shininess
delfv4 $light_position
```

In our interpreted interface, it is possible to interactively change parameters and see the effects of

various OpenGL functions. This a great way to figure out what various functions do and to try different things without having to recompile and run after each change.

Problems with the OpenGL interface

While the OpenGL interface we have generated is fully usable, it is not without problems.

- OpenGL constants are installed as global variables. As a result, it is necessary to use the global keyword when writing Tcl subroutines. For example :

```
proc clear_screan { } {
       global GL_COLOR_BUFFER_BIT, GL_DEPTH_BUFFER_BIT
       glClear $GL_COLOR_BUFFER_BIT
       glClear $GL_DEPTH_BUFFER_BIT
}
```

- Arrays need to be accessed via helper functions such as our `newfv4()` function. This approach certainly works and its easy enough to implement, but it may be preferable to call certain OpenGL functions with a Tcl list instead. For example :

```
glMaterialfv $GL_FRONT $GL_SPECULAR {1.0 1.0 1.0 1.0}
```

While these are only minor annoyances, it turns out that you can address both problems using SWIG typemaps (which are discussed shortly).

Exception handling

The `%except` directive can be used to create a user-definable exception handler in charge of converting exceptions in your C/C++ program into Tcl exceptions. The chapter on exception handling contains more details, but suppose you extended the array example into a C++ class like the following :

```
class RangeError {};   // Used for an exception

class DoubleArray {
  private:
    int n;
    double *ptr;
  public:
    // Create a new array of fixed size
    DoubleArray(int size) {
      ptr = new double[size];
      n = size;
    }
    // Destroy an array
    ~DoubleArray() {
       delete ptr;
    }
    // Return the length of the array
    int   length() {
      return n;
    }

    // Get an item from the array and perform bounds checking.
    double getitem(int i) {
```

```
    if ((i >= 0) && (i < n))
      return ptr[i];
    else
      throw RangeError();
  }

  // Set an item in the array and perform bounds checking.
  void setitem(int i, double val) {
    if ((i >= 0) && (i < n))
      ptr[i] = val;
    else {
      throw RangeError();
    }
  }
};
```

The functions associated with this class can throw a C++ range exception for an out-of-bounds array access. We can catch this in our Tcl extension by specifying the following in an interface file :

```
%except(tcl) {
  try {
    $function               // Gets substituted by actual function call
  }
  catch (RangeError) {
    interp->result = "Array index out-of-bounds";
    return TCL_ERROR;
  }
}
```

or in Tcl 8.0

```
%except(tcl8) {
  try {
    $function               // Gets substituted by actual function call
  }
  catch (RangeError) {
    Tcl_SetStringObj(tcl_result,"Array index out-of-bounds");
    return TCL_ERROR;
  }
}
```

When the C++ class throws a `RangeError` exception, our wrapper functions will catch it, turn it into a Tcl exception, and allow a graceful death as opposed to just having some sort of mysterious program crash. We are not limited to C++ exception handling. Please see the chapter on exception handling for more details on other possibilities, including a method for language-independent exception handling..

Typemaps

This section describes how SWIG's treatment of various C/C++ datatypes can be remapped using the `%typemap` directive. While not required, this section assumes some familiarity with Tcl's C API. The reader is advised to consult a Tcl book. A glance at the chapter on SWIG typemaps will also be useful.

What is a typemap?

A typemap is mechanism by which SWIG's processing of a particular C datatype can be changed. A simple typemap might look like this :

```
%module example

%typemap(tcl,in) int {
       $target = (int) atoi($source);
       printf("Received an integer : %d\n",$target);
}
...
extern int fact(int n);
```

Typemaps require a language name, method name, datatype, and conversion code. For Tcl, "tcl" should be used as the language name. For Tcl 8.0, "tcl8" should be used if you are using the native object interface. The "in" method in this example refers to an input argument of a function. The datatype 'int' tells SWIG that we are remapping integers. The supplied code is used to convert from a Tcl string to the corresponding C datatype. Within the supporting C code, the variable `$source` contains the source data (a string in this case) and `$target` contains the destination of a conversion (a C local variable).

When the example is compiled into a Tcl module, it will operate as follows :

```
% fact 6
Received an integer : 6
720
%
```

A full discussion of typemaps can be found in the main SWIG users reference. We will primarily be concerned with Tcl typemaps here.

Tcl typemaps

The following typemap methods are available to Tcl modules :

`%typemap(tcl,in)`	Converts a string to input function arguments
`%typemap(tcl,out)`	Converts return value of a C function to a string
`%typemap(tcl,freearg)`	Cleans up a function argument (if necessary)
`%typemap(tcl,argout)`	Output argument processing
`%typemap(tcl,ret)`	Cleanup of function return values
`%typemap(tcl,const)`	Creation of Tcl constants
`%typemap(memberin)`	Setting of C++ member data
`%typemap(memberout)`	Return of C++ member data
`%typemap(tcl, check)`	Check value of function arguments.

Typemap variables

The following variables may be used within the C code used in a typemap:

`$source`	Source value of a conversion
`$target`	Target of conversion (where the result should be stored)
`$type`	C datatype being remapped

`$mangle`	Mangled version of data (used for pointer type-checking)
`$value`	Value of a constant (const typemap only)
`$arg`	Original function argument (usually a string)

Name based type conversion

Typemaps are based both on the datatype and an optional name attached to a datatype. For example :

```
%module foo

// This typemap will be applied to all char ** function arguments
%typemap(tcl,in) char ** { ... }

// This typemap is applied only to char ** arguments named 'argv'
%typemap(tcl,in) char **argv { ... }
```

In this example, two typemaps are applied to the `char **` datatype. However, the second typemap will only be applied to arguments named '`argv`'. A named typemap will always override an unnamed typemap.

Due to the name-based nature of typemaps, it is important to note that typemaps are independent of typedef declarations. For example :

```
%typemap(tcl, in) double {
       ... get a double ...
}
void foo(double);           // Uses the above typemap
typedef double Real;
void bar(Real);             // Does not use the above typemap (double != Real)
```

To get around this problem, the `%apply` directive can be used as follows :

```
%typemap(tcl,in) double {
       ... get a double ...
}
void foo(double);

typedef double Real;        // Uses typemap
%apply double { Real };     // Applies all "double" typemaps to Real.
void bar(Real);             // Now uses the same typemap.
```

*Converting a Tcl list to a char ***

A common problem in many C programs is the processing of command line arguments, which are usually passed in an array of NULL terminated strings. The following SWIG interface file allows a Tcl list to be used as a `char **` object.

```
%module argv

// This tells SWIG to treat char ** as a special case
%typemap(tcl,in) char ** {
        int tempc;
        if (Tcl_SplitList(interp,$source,&tempc,&$target) == TCL_ERROR)
              return TCL_ERROR;
}
```

```
// This gives SWIG some cleanup code that will get called after the function call
%typemap(tcl,freearg) char ** {
        free((char *) $source);
}

// Return a char ** as a Tcl list
%typemap(tcl,out) char ** {
        int i = 0;
        while ($source[i]) {
             Tcl_AppendElement(interp,$source[i]);
             i++;
        }
}

// Now a few test functions
%inline %{
int print_args(char **argv) {
    int i = 0;
    while (argv[i]) {
         printf("argv[%d] = %s\n", i,argv[i]);
         i++;
    }
    return i;
}


// Returns a char ** list
char **get_args() {
    static char *values[] = { "Dave", "Mike", "Susan", "John", "Michelle", 0};
    return &values[0];
}

// A global variable
char *args[] = { "123", "54", "-2", "0", "NULL", 0 };

%}
%include tclsh.i
```

When compiled, we can use our functions as follows :

```
% print_args {John Guido Larry}
argv[0] = John
argv[1] = Guido
argv[2] = Larry
3
% puts [get_args]
Dave Mike Susan John Michelle
% puts [args_get]
123 54 -2 0 NULL
%
```

Perhaps the only tricky part of this example is the implicit memory allocation that is performed by the `Tcl_SplitList` function. To prevent a memory leak, we can use the SWIG “freearg” typemap to clean up the argument value after the function call is made. In this case, we simply free up the memory that `Tcl_SplitList` allocated for us.

Remapping constants

By default, SWIG installs C constants as Tcl read-only variables. Unfortunately, this has the undesirable side effect that constants need to be declared as "global" when used in subroutines. For example :

```
proc clearscreen { } {
       global GL_COLOR_BUFFER_BIT
       glClear $GL_COLOR_BUFFER_BIT
}
```

If you have hundreds of functions however, this quickly gets annoying. Here's a fix using hash tables and SWIG typemaps :

```
// Declare some Tcl hash table variables
%{
static Tcl_HashTable  constTable;      /* Hash table          */
static int           *swigconst;       /* Temporary variable  */
static Tcl_HashEntry *entryPtr;        /* Hash entry          */
static int            dummy;           /* dummy value         */
%}

// Initialize the hash table (This goes in the initialization function)

%init %{
  Tcl_InitHashTable(&constTable,TCL_STRING_KEYS);
%}

// A Typemap for creating constant values
// $source = the value of the constant
// $target = the name of the constant

%typemap(tcl,const) int, unsigned int, long, unsigned long {
  entryPtr = Tcl_CreateHashEntry(&constTable,"$target",&dummy);
  swigconst = (int *) malloc(sizeof(int));
  *swigconst = $source;
  Tcl_SetHashValue(entryPtr, swigconst);
  /* Make it so constants can also be used as variables */
  Tcl_LinkVar(interp,"$target", (char *) swigconst, TCL_LINK_INT | TCL_LINK_READ_ONLY);
};

// Now change integer handling to look for names in addition to values
%typemap(tcl,in) int, unsigned int, long, unsigned long {
  Tcl_HashEntry *entryPtr;
  entryPtr = Tcl_FindHashEntry(&constTable,$source);
  if (entryPtr) {
    $target = ($type) (*((int *) Tcl_GetHashValue(entryPtr)));
  } else {
    $target = ($type) atoi($source);
  }
}
```

In our Tcl code, we can now access constants by name without using the "global" keyword as follows :

```
proc clearscreen { } {
       glClear GL_COLOR_BUFFER_BIT
```

```
}
```

Returning values in arguments

The "argout" typemap can be used to return a value originating from a function argument. For example :

```
// A typemap defining how to return an argument by appending it to the result
%typemap(tcl,argout) double *outvalue {
        char dtemp[TCL_DOUBLE_SPACE];
        Tcl_PrintDouble(interp,*($source),dtemp);
        Tcl_AppendElement(interp, dtemp);
}

// A typemap telling SWIG to ignore an argument for input
// However, we still need to pass a pointer to the C function
%typemap(tcl,ignore) double *outvalue {
       static double temp;         /* A temporary holding place */
       $target = &temp;
}

// Now a function returning two values
int mypow(double a, double b, double *outvalue) {
        if ((a < 0) || (b < 0)) return -1;
        *outvalue = pow(a,b);
        return 0;
};
```

When wrapped, SWIG matches the `argout` typemap to the "`double *outvalue`" argument. The "ignore" typemap tells SWIG to simply ignore this argument when generating wrapper code. As a result, a Tcl function using these typemaps will work like this :

```
% mypow 2 3     # Returns two values, a status value and the result
0 8
%
```

An alternative approach to this is to return values in a Tcl variable as follows :

```
%typemap(tcl,argout) double *outvalue {
       char temp[TCL_DOUBLE_SPACE];
       Tcl_PrintDouble(interp,*($source),dtemp);
       Tcl_SetVar(interp,$arg,temp,0);
}
%typemap(tcl,in) double *outvalue {
       static double temp;
       $target = &temp;
}
```

Our Tcl script can now do the following :

```
% set status [mypow 2 3 a]
% puts $status
0
% puts $a
```

```
8.0
%
```

Here, we have passed the name of a Tcl variable to our C wrapper function which then places the return value in that variable. This is now very close to the way in which a C function calling this function would work.

Mapping C structures into Tcl Lists

Suppose you have a C structure like this :

```
typedef struct {
  char  login[16];          /* Login ID  */
  int   uid;                /* User ID   */
  int   gid;                /* Group ID  */
  char  name[32];           /* User name */
  char  home[256];          /* Home directory */
} User;
```

By default, SWIG will simply treat all occurrences of "`User`" as a pointer. Thus, functions like this :

```
extern void add_user(User u);
extern User *lookup_user(char *name);
```

will work, but they will be weird. In fact, they may not work at all unless you write helper functions to create users and extract data. A typemap can be used to fix this problem however. For example :

```
// This works for both "User" and "User *"
%typemap(tcl,in) User * {
       int tempc;
       char **tempa;
       static User temp;
       if (Tcl_SplitList(interp,$source,&tempc,&tempa) == TCL_ERROR) return TCL_ERROR;
       if (tempc != 5) {
              free((char *) tempa);
              interp->result = "Not a valid User record";
              return TCL_ERROR;
       }
       /* Split out the different fields */
       strncpy(temp.login,tempa[0],16);
       temp.uid = atoi(tempa[1]);
       temp.gid = atoi(tempa[2]);
       strncpy(temp.name,tempa[3],32);
       strncpy(temp.home,tempa[4],256);
       $target = &temp;
       free((char *) tempa);
}

// Describe how we want to return a user record
%typemap(tcl,out) User * {
       char temp[20];
       if ($source) {
       Tcl_AppendElement(interp,$source->login);
       sprintf(temp,"%d",$source->uid);
       Tcl_AppendElement(interp,temp);
       sprintf(temp,"%d",$source->gid);
```

```
            Tcl_AppendElement(interp,temp);
            Tcl_AppendElement(interp,$source->name);
            Tcl_AppendElement(interp,$source->home);
            }
}
```

These function marshall Tcl lists to and from our `User` data structure. This allows a more natural implementation that we can use as follows :

```
% add_user {beazley 500 500 "Dave Beazley" "/home/beazley"}
% lookup_user beazley
beazley 500 500 {Dave Beazley} /home/beazley
```

This is a much cleaner interface (although at the cost of some performance). The only caution I offer is that the pointer view of the world is pervasive throughout SWIG. Remapping complex datatypes like this will usually work, but every now and then you might find that it breaks. For example, if we needed to manipulate arrays of Users (also mapped as a "`User *`"), the typemaps defined here would break down and something else would be needed. Changing the representation in this manner may also break the object-oriented interface.

Useful functions

The following tables provide some functions that may be useful in writing Tcl typemaps. Both Tcl 7.x and Tcl 8.x are covered. For Tcl 7.x, everything is a string so the interface is relatively simple. For Tcl 8, everything is now a Tcl object so a more precise set of functions is required. Given the alpha-release status of Tcl 8, the functions described here may change in future releases.

Tcl 7.x Numerical Conversion Functions

`int Tcl_GetInt(Tcl_Interp *,char *, int *ip)`	Convert a string to an integer which is stored in `ip`. Returns TCL_OK on success, TCL_ERROR on failure.
`int Tcl_GetDouble(Tcl_Interp *, char *, double *dp)`	Convert a string to a double which is stored in `*dp`. Returns TCL_OK on success, TCL_ERROR on failure.
`Tcl_PrintDouble(Tcl_Interp *, double val, char *dest)`	Creates a string with a double precision value. The precision is determined by the value of the `tcl_precision` variable.

Tcl 7.x String and List Manipulation Functions

`void Tcl_SetResult(Tcl_Interp *, char *str, Tcl_FreeProc *freeProc)`	Set the Tcl result string. `str` is the string and `freeProc` is a procedure to free the result. This is usually TCL_STATIC, TCL_DYNAMIC, TCL_VOLATILE.
`void Tcl_AppendResult(Tcl_Interp *, char *str, char *str, ... (char *) NULL)`	Append string elements to the Tcl result string.

Tcl 7.x String and List Manipulation Functions

`void Tcl_AppendElement(Tcl_Interp *, char *string)`	Formats string as a Tcl list and appends it to the result string.
`int Tcl_SplitList(Tcl_Interp *, char *list, int *argcPtr, char ***argvPtr)`	Parses list as a Tcl list and creates an array of strings. The number of elements is stored in `*argcPtr`. A pointer to the string array is stored in `***argvPtr`. Returns TCL_OK on success, TCL_ERROR if an error occurred. The pointer value stored in `argvPtr` must eventual be passed to `free`().
`char *Tcl_Merge(int argc, char **argv)`	The inverse of SplitList. Returns a pointer to a Tcl list that has been formed from the array `argv`. The result is dynamically allocated and must be passed to free by the caller.

Tcl 8.x Integer Conversion Functions

`Tcl_Obj *Tcl_NewIntObj(int Value)`	Create a new integer object.
`void Tcl_SetIntObj(Tcl_Obj *obj, int Value)`	Set the value of an integer object
`int Tcl_GetIntFromObj(Tcl_Interp *, Tcl_Obj *obj, int *ip)`	Get the integer value of an object and return it in `*ip`. Returns TCL_ERROR if the object is not an integer.

Tcl 8.x Floating Point Conversion Functions

`Tcl_Obj *Tcl_NewDoubleObj(double value)`	Create a new Tcl object containing a double.
`Tcl_SetDoubleObj(Tcl_Obj *obj, double value)`	Set the value of a `Tcl_Object`.
`int Tcl_GetDoubleFromObj(Tcl_Interp, Tcl_Obj *o, double *dp)`	Get a double from a Tcl object. The value is stored in `*dp`. Returns TCL_OK on success, TCL_ERROR if the conversion can't be made.

Tcl 8.x String Conversion Functions

`Tcl_Obj *Tcl_NewStringObj(char *str, int len)`	Creates a new Tcl string object. `str` contains the ASCII string, `len` contains the number of bytes or -1 if the string is NULL terminated.
`Tcl_SetStringObj(Tcl_Obj *obj, char *str, int len)`	Sets a Tcl object to a given string. `len` is the string length or -1 if the string is NULL terminated.

Tcl 8.x String Conversion Functions

`char *Tcl_GetStringFromObj(Tcl_Obj *obj, int *len)`	Retrieves the string associated with an object. The length is returned in `*len`;
`Tcl_AppendToObj(Tcl_Obj *obj, char *str, int len)`	Appends the string `str` to the given Tcl Object. `len` contains the number of bytes of -1 if NULL terminated.

Tcl 8.x List Conversion Functions

`Tcl_Obj *Tcl_NewListObj(int objc, Tcl_Obj *objv)`	Creates a new Tcl List object. `objc` contains the element count and `objv` is an array of Tcl objects.
`int Tcl_ListObjAppendList(Tcl_Interp *, Tcl_Obj *listPtr, Tcl_Obj *elemListPtr)`	Appends the objects in `elemListPtr` to the list object `listPtr`. Returns TCL_ERROR if an error occurred.
`int Tcl_ListObjAppendElement(Tcl_Interp *, Tcl_Obj *listPtr, Tcl_Obj *element)`	Appends element to the end of the list object `listPtr`. Returns TCL_ERROR if an error occurred. Will convert the object pointed to by listPtr to a list if it isn't one already.
`int Tcl_ListObjGetElements(Tcl_Interp *, Tcl_Obj *listPtr, int *objcPtr, Tcl_Obj ***objvPtr)`	Converst a Tcl List object into an array of pointers to individual elements. `objcPtr` receives the list length and `objvPtr` receives a pointer to an array of Tcl_Obj pointers. Returns TCL_ERROR if the list can not be converted.
`int Tcl_ListObjLength(Tcl_Interp *, Tcl_Obj *listPtr, int *intPtr)`	Returns the length of a list in `intPtr`.If the object is not a list or an error occurs, the function returns TCL_ERROR.
`int Tcl_ListObjIndex(Tcl_Interp *, Tcl_Obj *listPtr, int index, Tcl_Obj **objptr)`	Returns the pointer to object with given index in the list. Returns TCL_ERROR if `listPtr` is not a list or the index is out of range. The pointer is returned in `objptr`.
`int Tcl_ListObjReplace(Tcl_Interp *, Tcl_Obj *listPtr, int first, int count, int objc, Tcl_Obj *objv)`	Replaces objects in a list. `first` is the first object to replace and `count` is the total number of objects. `objc` and `objv` define a set of new objects to insert into the list. If `objv` is NULL, no new objects will be added and the function acts as a deletion operation.

Tcl 8.x Object Manipulation

`Tcl_Obj *Tcl_NewObj()`	Create a new Tcl object
`Tcl_Obj *Tcl_DuplicateObj(Tcl_Obj *obj)`	Duplicate a Tcl object.
`Tcl_IncrRefCount(Tcl_Obj *obj)`	Increase the reference count on an object.
`Tcl_DecrRefCount(Tcl_Obj *obj)`	Decrement the reference count on an object.
`int Tcl_IsShared(Tcl_Obj *obj)`	Tests to see if an object is shared.

Standard typemaps

The following typemaps show how to convert a few common kinds of objects between Tcl and C (and to give a better idea of how typemaps work)

Function argument typemaps (Tcl 7.x)

`int, short, long`	`%typemap(tcl,in) int,short,long {`
	`  int temp;`
	`  if (Tcl_GetInt(interp,$source,&temp) == TCL_ERROR)`
	`  return TCL_ERROR;`
	`  $target = ($type) temp;`
	`}`

`float, double`	`%typemap(tcl,in) double,float {`
	`  double temp;`
	`  if (Tcl_GetDouble(interp,$source,&temp)`
	`  == TCL_ERROR)`
	`  return TCL_ERROR;`
	`  $target = ($type) temp;`
	`}`

`char *`	`%typemap(tcl,in) char * {`
	`  $target = $source;`
	`}`

Function return typemaps (Tcl 7.x)

`int, short, long,`	`%typemap(tcl,out) int, short, long {`
	`  sprintf($target,"%ld", (long) $source);`
	`}`

`float, double`	`%typemap(tcl,out) float,double {`
	`  Tcl_PrintDouble(interp,$source,interp->result);`
	`}`

Function return typemaps (Tcl 7.x)

char *	%typemap(tcl,out) char * {
	  Tcl_SetResult(interp,$source,TCL_VOLATILE);
	}

Function argument typemaps (Tcl 8.x)

int,
short,
long
	%typemap(tcl8,in) int,short,long {
	  int temp;
	  if (Tcl_GetIntFromObj(interp,$source,&temp) ==
	  TCL_ERROR)
	  return TCL_ERROR;
	  $target = ($type) temp;
	}

float,
double
	%typemap(tcl8,in) double,float {
	  double temp;
	  if (Tcl_GetDoubleFromObj(interp,$source,&temp)
	  == TCL_ERROR)
	  return TCL_ERROR;
	  $target = ($type) temp;
	}

char *	%typemap(tcl8,in) char * {
	  int len;
	  $target = Tcl_GetStringFromObj(interp,&len);
	}

Function return typemaps (Tcl 8.x)

int,
short,
long,
	%typemap(tcl8,out) int, short, long {
	  Tcl_SetIntObj($target,$source);
	}

float,
double
	%typemap(tcl8,out) float,double {
	  Tcl_SetDoubleObj($target,$source);
	}

char *	%typemap(tcl8,out) char * {
	  Tcl_SetStringObj($target,$source)
	}

Pointer handling

SWIG pointers are mapped into Python strings containing the hexadecimal value and type. The following functions can be used to create and read pointer values .

SWIG Pointer Conversion Functions (Tcl 7.x/8.x)

`void SWIG_MakePtr(char *str, void *ptr, char *type)`
 	
`void SWIG_SetPointerObj(Tcl_Obj *objPtr, void *ptr, char *type)`
	Makes a pointer string and saves it in `str`, which must be large enough to hold the result. `ptr` contains the pointer value and `type` is the string representation of the type.

`char *SWIG_GetPtr(char *str, void **ptr, char *type)`
 	
`char *SWIG_GetPointerObj(Tcl_Interp *interp, Tcl_Obj *objPtr, void **ptr, char *_t)`
	Attempts to read a pointer from the string `str`. `ptr` is the address of the pointer to be created and `type` is the expected type. If `type` is NULL, then any pointer value will be accepted. On success, this function returns NULL. On failure, it returns the pointer to the invalid portion of the pointer string.

These functions can be used in typemaps as well. For example, the following typemap makes an argument of "`char *buffer`" accept a pointer instead of a NULL-terminated ASCII string.

```
%typemap(tcl,in) char *buffer {
       if (SWIG_GetPtr($source, (void **) &$target, "$mangle")) {
              Tcl_SetResult(interp,"Type error. Not a pointer", TCL_STATIC);
              return TCL_ERROR;
       }
}
```

Note that the `$mangle` variable generates the type string associated with the datatype used in the typemap.

By now you hopefully have the idea that typemaps are a powerful mechanism for building more specialized applications. While writing typemaps can be technical, many have already been written for you. See the SWIG library reference for more information.

Configuration management with SWIG

After you start to work with Tcl for awhile, you suddenly realize that there are an unimaginable number of extensions, tools, and other packages. To make matters worse, there are about 20 billion different versions of Tcl, not all of which are compatible with each extension (this is to make life interesting of course).

While SWIG is certainly not a magical solution to the configuration management problem, it can help out alot in a number of key areas :

- SWIG generated code can be used with all versions of Tcl/Tk newer than 7.3/3.6. This includes the Tcl Netscape Plugin and Tcl 8.0a2.
- The SWIG library mechanism can be used to manage various code fragments and initialization functions.
- SWIG generated code usually requires no modification so it is relatively easy to switch

between different Tcl versions as necessary or upgrade to a newer version when the time comes (of course, the Sun Tcl/Tk team might have changed other things to keep you occupied)

Writing a main program and Tcl_AppInit()

The traditional method of creating a new Tcl extension required a programmer to write a special function called `Tcl_AppInit()` that would initialize your extension and start the Tcl interpreter. A typical `Tcl_AppInit()` function looks like the following :

```
/* main.c */
#include <tcl.h>

main(int argc, char *argv[]) {
       Tcl_Main(argc,argv);
       exit(0);
}

int Tcl_AppInit(Tcl_Interp *interp) {
       if (Tcl_Init(interp) == TCL_ERROR) {
              return TCL_ERROR;
       }

       /* Initialize your extension */
       if (Your_Init(interp) == TCL_ERROR) {
              return TCL_ERROR;
       }

       tcl_RcFileName = "~/.myapp.tcl";
       return TCL_OK;
}
```

While relatively simple to write, there are tons of problems with doing this. First, each extension that you use typically has their own `Tcl_AppInit()` function. This forces you to write a special one to initialize everything by hand. Secondly, the process of writing a main program and initializing the interpreter varies between different versions of Tcl and different platforms. For example, in Tcl 7.4, the variable "`tcl_RcFileName`" is a C variable while in Tcl7.5 and newer versions its a Tcl variable instead. Similarly, the `Tcl_AppInit` function written for a Unix machine might not compile correctly on a Mac or Windows machine.

In SWIG, it is almost never necessary to write a `Tcl_AppInit()` function because this is now done by SWIG library files such as `tclsh.i` or `wish.i`. To give a better idea of what these files do, here's the code from the SWIG `tclsh.i` file which is roughly comparable to the above code

```
// tclsh.i : SWIG library file for rebuilding tclsh
%{

/* A TCL_AppInit() function that lets you build a new copy
 * of tclsh.
 *
 * The macro SWIG_init contains the name of the initialization
 * function in the wrapper file.
 */
```

```
#ifndef SWIG_RcFileName
char *SWIG_RcFileName = "~/.myapprc";
#endif

int Tcl_AppInit(Tcl_Interp *interp){

  if (Tcl_Init(interp) == TCL_ERROR)
    return TCL_ERROR;

  /* Now initialize our functions */
  if (SWIG_init(interp) == TCL_ERROR)
    return TCL_ERROR;

#if TCL_MAJOR_VERSION > 7 || TCL_MAJOR_VERSION == 7 && TCL_MINOR_VERSION >= 5
   Tcl_SetVar(interp,"tcl_rcFileName",SWIG_RcFileName,TCL_GLOBAL_ONLY);
#else
   tcl_RcFileName = SWIG_RcFileName;
#endif
  return TCL_OK;
}

#if TCL_MAJOR_VERSION > 7 || TCL_MAJOR_VERSION == 7 && TCL_MINOR_VERSION >= 4
int main(int argc, char **argv) {
  Tcl_Main(argc, argv, Tcl_AppInit);
  return(0);

}
#else
extern int main();
#endif

%}
```

This file is essentially the same as a normal `Tcl_AppInit()` function except that it supports a variety of Tcl versions. When included into an interface file, the symbol `SWIG_init` contains the actual name of the initialization function (This symbol is defined by SWIG when it creates the wrapper code). Similarly, a startup file can be defined by simply defining the symbol `SWIG_RcFileName`. Thus, a typical interface file might look like this :

```
%module graph
%{
#include "graph.h"
#define SWIG_RcFileName "graph.tcl"
%}

%include tclsh.i

... declarations ...
```

By including the `tclsh.i`, you automatically get a `Tcl_AppInit()` function. A variety of library files are also available. `wish.i` can be used to build a new wish executable, `expect.i` contains the main program for Expect, and `ish.i`, `itclsh.i`, `iwish.i`, and `itkwish.i` contain initializations for various incarnations of [incr Tcl].

Creating a new package initialization library

If a particular Tcl extension requires special initialization, you can create a special SWIG library file to initialize it. For example, a library file to extend Expect looks like the following :

```
// expect.i : SWIG Library file for Expect
%{

/* main.c - main() and some logging routines for expect

Written by: Don Libes, NIST, 2/6/90

Design and implementation of this program was paid for by U.S. tax
dollars.  Therefore it is public domain.  However, the author and NIST
would appreciate credit if this program or parts of it are used.
*/

#include "expect_cf.h"
#include <stdio.h>
#include INCLUDE_TCL
#include "expect_tcl.h"

void
main(argc, argv)
int argc;
char *argv[];
{
        int rc = 0;
        Tcl_Interp *interp = Tcl_CreateInterp();
        int SWIG_init(Tcl_Interp *);

        if (Tcl_Init(interp) == TCL_ERROR) {
                fprintf(stderr,"Tcl_Init failed: %s\n",interp->result);
                exit(1);
        }
        if (Exp_Init(interp) == TCL_ERROR) {
                fprintf(stderr,"Exp_Init failed: %s\n",interp->result);
                exit(1);
        }

        /* SWIG initialization. --- 2/11/96  */
        if (SWIG_init(interp) == TCL_ERROR) {
                fprintf(stderr,"SWIG initialization failed: %s\n", interp->result);
                exit(1);
        }

        exp_parse_argv(interp,argc,argv);
        /* become interactive if requested or "nothing to do" */
        if (exp_interactive)
                (void) exp_interpreter(interp);
        else if (exp_cmdfile)
                rc = exp_interpret_cmdfile(interp,exp_cmdfile);
        else if (exp_cmdfilename)
                rc = exp_interpret_cmdfilename(interp,exp_cmdfilename);

        /* assert(exp_cmdlinecmds != 0) */
        exp_exit(interp,rc);
```

```
        /*NOTREACHED*/
}
%}
```

In the event that you need to write a new library file such as this, the process usually isn't too difficult. Start by grabbing the original `Tcl_AppInit()` function for the package. Enclose it in a `%{,%}` block. Now add a line that makes a call to `SWIG_init()`. This will automatically resolve to the real initialization function when compiled.

Combining Tcl/Tk Extensions

A slightly different problem concerns the mixing of various extensions. Most extensions don't require any special initialization other than calling their initialization function. To do this, we also use SWIG library mechanism. For example :

```
// blt.i : SWIG library file for initializing the BLT extension
%{
#ifdef __cplusplus
extern "C" {
#endif
extern int Blt_Init(Tcl_Interp *);
#ifdef __cplusplus
}
#endif
%}
%init %{
       if (Blt_Init(interp) == TCL_ERROR) {
              return TCL_ERROR;
       }
%}


// tix.i : SWIG library file for initializing the Tix extension
%{
#ifdef __cplusplus
extern "C" {
#endif
extern int Tix_Init(Tcl_Interp *);
#ifdef __cplusplus
}
#endif
%}
%init %{
       if (Tix_Init(interp) == TCL_ERROR) {
              return TCL_ERROR;
       }
%}
```

Both files declare the proper initialization function (to be C++ friendly, this should be done using `extern "C"`). A call to the initialization function is then placed inside a `%init %{ ... %}` block.

To use our library files and build a new version of wish, we might now do the following :

```
// mywish.i : wish with a bunch of stuff added to it
%include wish.i
```

```
%include blt.i
%include tix.i

... additional declarations ...
```

Of course, the really cool part about all of this is that the file '`mywish.i`' can itself, serve as a library file. Thus, when building various versions of Tcl, we can place everything we want to use a special file and use it in all of our other interface files :

```
// interface.i
%module mymodule

%include mywish.i              // Build our version of Tcl with extensions

... C declarations ...
```

or we can grab it on the command line :

```
unix > swig -tcl -lmywish.i interface.i
```

Limitations to this approach

This interface generation approach is limited by the compatibility of each extension you use. If any one extension is incompatible with the version of Tcl you are using, you may be out of luck. It is also critical to pay careful attention to libraries and include files. An extension library compiled against an older version of Tcl may fail when linked with a newer version.

Dynamic loading

Newer versions of Tcl support dynamic loading. With dynamic loading, you compile each extension into a separate module that can be loaded at run time. This simplifies a number of compilation and extension building problems at the expense of creating new ones. Most notably, the dynamic loading process varies widely between machines and is not even supported in some cases. It also does not work well with C++ programs that use static constructors. Modules linked with older versions of Tcl may not work with newer versions as well (although SWIG only really uses the basic Tcl C interface). As a result, I usually find myself using both dynamic and static linking as appropriate.

Turning a SWIG module into a Tcl Package.

Tcl 7.4 introduced the idea of an extension package. By default, SWIG does not create "packages", but it is relatively easy to do. To make a C extension into a Tcl package, you need to provide a call to `Tcl_PkgProvide()` in the module initialization function. This can be done in an interface file as follows :

```
%init %{
        Tcl_PkgProvide(interp,"example","0.0");
%}
```

Where "example" is the name of the package and "0.0" is the version of the package.

Next, after building the SWIG generated module, you need to execute the "`pkg_mkIndex`" command inside tclsh. For example :

```
unix > tclsh
% pkg_mkIndex . example.so
% exit
```

This creates a file "`pkgIndex.tcl`" with information about the package. To use your package, you now need to move it to its own subdirectory which has the same name as the package. For example :

```
./example/
          pkgIndex.tcl             # The file created by pkg_mkIndex
          example.so               # The SWIG generated module
```

Finally, assuming that you're not entirely confused at this point, make sure that the example subdirectory is visible from the directories contained in either the `tcl_library` or `auto_path` variables. At this point you're ready to use the package as follows :

```
unix > tclsh
% package require example
% fact 4
24
%
```

If you're working with an example in the current directory and this doesn't work, do this instead :

```
unix > tclsh
% lappend auto_path .
% package require example
% fact 4
24
```

As a final note, most SWIG examples do not yet use the `package` commands. For simple extensions it may be easier just to use the `load` command instead.

Building new kinds of Tcl interfaces (in Tcl)

One of the most interesting aspects of Tcl and SWIG is that you can create entirely new kinds of Tcl interfaces in Tcl using the low-level SWIG accessor functions. For example, suppose you had a library of helper functions to access arrays :

```
/* File : array.i */
%module array

%inline %{
double *new_double(int size) {
        return (double *) malloc(size*sizeof(double));
}
void delete_double(double *a) {
        free(a);
}
double get_double(double *a, int index) {
```

```
        return a[index];
}
void set_double(double *a, int index, double val) {
        a[index] = val;
}
int *new_int(int size) {
        return (int *) malloc(size*sizeof(int));
}
void delete_int(int *a) {
        free(a);
}
int get_int(int *a, int index) {
        return a[index];
}
int set_int(int *a, int index, int val) {
        a[index] = val;
}
%}
```

While these could be called directly, we could also write a Tcl script like this :

```
proc Array {type size} {
    set ptr [new_$type $size]
    set code {
        set method [lindex $args 0]
        set parms [concat $ptr [lrange $args 1 end]]
        switch $method {
            get {return [eval "get_$type $parms"]}
            set {return [eval "set_$type $parms"]}
            delete {eval "delete_$type $ptr; rename $ptr {}"}
        }
    }
    # Create a procedure
    uplevel "proc $ptr args {set ptr $ptr; set type $type;$code}"
    return $ptr
}
```

Our script allows easy array access as follows :

```
set a [Array double 100]                    ;# Create a double [100]
for {set i 0} {$i < 100} {incr i 1} {       ;# Clear the array
       $a set $i 0.0
}
$a set 3 3.1455                             ;# Set an individual element
set b [$a get 10]                           ;# Retrieve an element

set ia [Array int 50]                       ;# Create an int[50]
for {set i 0} {$i < 50} {incr i 1} {        ;# Clear it
       $ia set $i 0
}
$ia set 3 7                                 ;# Set an individual element
set ib [$ia get 10]                         ;# Get an individual element

$a delete                                   ;# Destroy a
$ia delete                                  ;# Destroy ia
```

The cool thing about this approach is that it makes a common interface for two different types of

arrays. In fact, if we were to add more C datatypes to our wrapper file, the Tcl code would work with those as well--without modification. If an unsupported datatype was requested, the Tcl code would simply return with an error so there is very little danger of blowing something up (although it is easily accomplished with an out of bounds array access).

Shadow classes

A similar approach can be applied to shadow classes. The following example is provided by Erik Bierwagen and Paul Saxe. To use it, run SWIG with the `-noobject` option (which disables the builtin object oriented interface). When running Tcl, simply source this file. Now, objects can be used in a more or less natural fashion.

```
# swig_c++.tcl
# Provides a simple object oriented interface using
# SWIG's low level interface.
#

proc new {objectType handle_r args} {
    # Creates a new SWIG object of the given type,
    # returning a handle in the variable "handle_r".
    #
    # Also creates a procedure for the object and a trace on
    # the handle variable that deletes the object when the
    # handle varibale is overwritten or unset
    upvar $handle_r handle
    #
    # Create the new object
    #
    eval set handle \[new_$objectType $args\]
    #
    # Set up the object procedure
    #
    proc $handle {cmd args} "eval ${objectType}_\$cmd $handle \$args"
    #
    # And the trace ...
    #
    uplevel trace variable $handle_r uw "{deleteObject $objectType $handle}"
    #
    # Return the handle so that 'new' can be used as an argument to a procedure
    #
    return $handle
}

proc deleteObject {objectType handle name element op} {
    #
    # Check that the object handle has a reasonable form
    #
    if {![regexp {_[0-9a-f]*_(.+)_p} $handle]} {
        error "deleteObject: not a valid object handle: $handle"
    }
    #
    # Remove the object procedure
    #
    catch {rename $handle {}}
    #
    # Delete the object
    #
```

```
    delete_$objectType $handle
}

proc delete {handle_r} {
    #
    # A synonym for unset that is more familiar to C++ programmers
    #
    uplevel unset $handle_r
}
```

To use this file, we simply source it and execute commands such as “new” and “delete” to manipulate objects. For example :

```
// list.i
%module List
%{
#include "list.h"
%}

// Very simple C++ example

class List {
public:
  List();  // Create a new list
  ~List(); // Destroy a list
  int  search(char *value);
  void insert(char *);  // Insert a new item into the list
  void remove(char *);  // Remove item from list
  char *get(int n);     // Get the nth item in the list
  int  length;          // The current length of the list
static void print(List *l);  // Print out the contents of the list
};
```

Now a Tcl script using the interface...

```
load ./list.so list       ; # Load the module
source swig_c++.tcl       ; # Source the object file

new List l
$l insert Dave
$l insert John
$l insert Guido
$l remove Dave
puts $l length_get

delete l
```

The cool thing about this example is that it works with any C++ object wrapped by SWIG and requires no special compilation. Proof that a short, but clever Tcl script can be combined with SWIG to do many interesting things.

Extending the Tcl Netscape Plugin

SWIG can be used to extend the Tcl Netscape plugin with C functions. As of this writing it has only been tested with version 1.0 of the plugin on Solaris and Irix 6.2. It may work on other

machines as well. However, first a word of caution --- doing this might result in serious injury as you can add just about any C function you want. Furthermore, it's not portable (hey, we're talking C code here). It seems like the best application of this would be creating a browser interface to a highly specialized application. Any scripts that you would write would not work on other machines unless they also installed the C extension code as well. Perhaps we should call this a plugin-plugin...

To use the plugin, use the `-plugin` option :

```
swig -tcl -plugin interface.i
```

This adds a "safe" initialization function compatible with the plugin (in reality, it just calls the function SWIG already creates). You also need to put the following symbol in your interface file for it to work :

```
%{
#define SAFE_SWIG
%}
```

The folks at Sun are quite concerned about the security implications of this sort of extension and originally wanted the user to modify the wrapper code by hand to "remind" them that they were installing functions into a safe interpreter. However, having seen alot of SWIG generated wrapper code, I hated that idea (okay, so the output of SWIG is just a little messy). This is compromise--you need to put that `#define` into your C file someplace. You can also just make it a compiler option if you would like.

The step-by-step process for making a plugin extension.

Making a plugin extension is relatively straightforward but you need to follow these steps :

- Make sure you have Tcl7.6/Tk4.2 installed on your machine. We're going to need the header files into order to compile the extension.
- Make sure you have the Netscape plugin properly installed.
- Run SWIG using the '`-tcl -plugin`' options.
- Compile the extension using the Tcl 7.6/Tk4.2 header files, but linking against the plugin itself. For example :

```
unix > gcc -I/usr/local/include -c example.o interface_wrap.c
unix > ld -shared example.o interface_wrap.o \
       -L/home/beazley/.netscape/plugins/libtclplugin.so -o example.so
```

- Copy the shared object file to the `~/.tclplug/tcl7.7` directory.

Using the plugin

To use the plugin, place the following line in your Tcl scripts :

```
load $tcl_library/example.so example
```

With luck, you will now be ready to run (at least that's the theory).

Tcl8.0 features

SWIG 1.1 now supports Tcl 8.0. However, considering the beta release nature of Tcl 8.0, anything presented here is subject to change. Currently only Tcl 8.0b1 is supported. None of the alpha releases are supported due to a change in the C API.

The Tcl 8.0 module uses the new Tcl 8.0 object interface whenever possible. Instead of using strings, the object interface provides more direct access to objects in their native representation. As a result, the performance is significantly better. The older Tcl SWIG module is also compatible with Tcl 8.0, but since it uses strings it will be much slower than the new version.

In addition to using native Tcl objects, the Tcl8.0 manipulates pointers directly in in a special Tcl object. On the surface it still looks like a string, but internally its represented a (value,type) pair. This too, should offer somewhat better performance.