Extending SWIG 12

Introduction

This chapter attempts to describe the process of extending SWIG to support new target languages and documentation methods. First a word of warning--SWIG started out being a relatively simple system for building interfaces to ANSI C programs. Since then, it has grown into something much more than that (although I'm still trying to figure out what). As a result, it is undergoing a number of growing pains. Certain parts of the code have been rewritten and others can probably be described as a hackish nightmare. I'm always working on ways to improve the implementation, but expect to find a few warts and inconsistencies.

Prerequisites

In order to develop or modify a SWIG module, I assume the following :

- That you understand the C API for the scripting language of interest.
- You have a good understanding of how SWIG operates and a good idea of how typemaps work.
- That you have some experience with C++. SWIG is written in C++, but doesn't use it maximally. However, familiarity with classes, inheritance, and operator overloading will help.
- That you're just a little crazy (this will help alot).

SWIG Organization

SWIG is built around a central core of functions and classes responsible for parsing interface files, managing documentation, handling datatypes, and utility functions. This code is contained in the "SWIG" directory of the distribution, but contains no information specific to any one scripting language. The various scripting language modules are implemented as C++ classes and found in the "Modules" directory of the distribution. The basic idea behind writing a module is that you write a language class containing about a dozen methods for creating wrapper functions, variables, constants, etc.... To use the language, you simply create a language "object", pass it on the parser and you magically get wrapper code. Documentation modules are written in a similar manner.

An important aspect of the design is the relationship between ANSI C and C++. The original version of SWIG was developed to support ANSI C programs. To add C++ support, an additional "layer" was added to the system---that is, all of the C++ support is really built on top of the ANSI C support. Language modules can take advantage of both C and C++ although a module written only for C can still work with C++ (due to the layered implementation).

As for making modifications to SWIG, all files in the "SWIG" directory should be considered "critical." Making changes here can cause serious problems in all SWIG language modules. When making customizations, one should only consider files in the "Modules" directory if at all possible.

The organization of this chapter

The remainder of this chapter is a bottom-up approach is to building SWIG modules. It will start with the basics and gradually build up a working language module, introducing new concepts as needed.

Compiling a SWIG extension

The first order of business is that of compiling an extension to SWIG and using it. This is the easy part.

Required files

To build any SWIG extension you need to locate the files "`swig.h`" and "`libswig.a`". In a typical installation, these will usually be found in `/usr/local/include` and `/usr/local/lib` respectively. All extension modules will need to include the "`swig.h`" header file and link against the `libswig.a` library.

Required C++ compiler

Due to name-mangling in the C++ compiler (which is different between compiler implementations), you will need to use the same C++ compiler used to compile SWIG. If you don't know which C++ compiler was used, typing '`swig -version`' will cause SWIG to print out its version number and the C++ compiler that was used to build it.

Writing a main program

To get any extension to work, it is necessary to write a small `main()` program to create a language object and start the SWIG parser. For example :

```
#include <swig.h>
#include "swigtcl.h"                        // Language specific header

extern int SWIG_main(int, char **, Language *, Documentation *);
int main(int argc, char **argv) {

       TCL *l = new Tcl;                    // Create a new Language object
       init_args(argc, argv);               // Initialize args
       return SWIG_main(argc, argv, l, 0);
}
```

Compiling

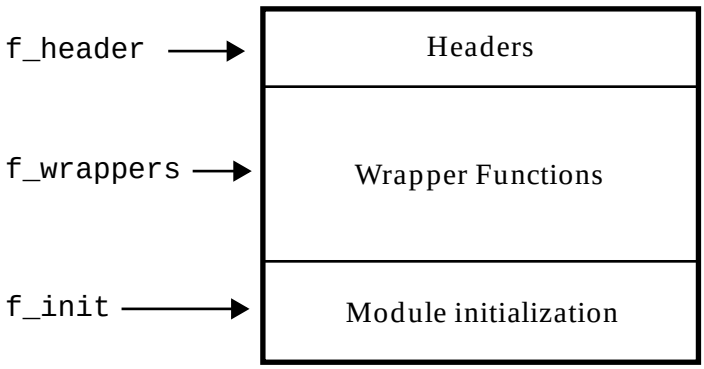
To compile your extension, do the following :

```
% c++ tcl.cxx main.cxx -lswig -o myswig
```

In this case we get a special version of SWIG that compiles Tcl extensions.

SWIG output

The output of SWIG is a single file that is organized as follows :



During code generation, the three sections are created as separate files that are accessed using the following file handles :

```
FILE *f_header;             // Header section
FILE *f_wrappers;           // Wrapper section
FILE *f_init;               // Initialization function
```

On exit, the three files are merged into a single output file.

When generating code, your language module should use the I/O functions in the C `<stdio.h>` library. SWIG does not use the C++ streams library.

The use of each output section can be roughly described as follows :

- The header section contains forward declarations, header files, helper functions, and run-time functions (such as the pointer type-checker). All code included with %{,%} also ends up here.
- The wrapper section contains all of the SWIG generated wrapper functions.
- The initialization section is a single C function used to initialize the module. For large modules, this function can be quite large. In any case, output to `f_init` should be treated with some care considering that the file is essentially one big C function.

The Language class (simple version)

Writing a new language module involves inheriting from the SWIG `Language` class and implementing methods for a few virtual functions. A minimal definition of a new Language module is as follows :

```
// File : mylang.h
// A minimal SWIG Language module

class MYLANG : public Language {
private:
       char *module;
```

```
public :
       MYLANG() {
              module = 0;
       };
       // Virtual functions required by the SWIG parser
        void parse_args(int, char *argv[]);
        void parse();
        void create_function(char *, char *, DataType *, ParmList *);
        void link_variable(char *, char *, DataType *);
        void declare_const(char *, char *, DataType *, char *);
        void initialize(void);
        void headers(void);
        void close(void);
        void set_module(char *,char **);
        void create_command(char *, char *);
};
```

Given the above header file, we can create a very simplistic language module as follows :

```
// -----------------------------------------------------------------------
// A simple SWIG Language module
// -----------------------------------------------------------------------

#include "swig.h"
#include "mylang.h"

// -----------------------------------------------------------------------
// MYLANG::parse_args(int argc, char *argv[])
//
// Parse command line options and initializes variables.
// -----------------------------------------------------------------------
void MYLANG::parse_args(int argc, char *argv[]) {
       printf("Getting command line options\n");
       typemap_lang = "mylang";
}

// -----------------------------------------------------------------------
// void MYLANG::parse()
//
// Start parsing an interface file.
// -----------------------------------------------------------------------
void MYLANG::parse() {
       fprintf(stderr,"Making wrappers for My Language\n");
       headers();
       yyparse();       // Run the SWIG parser
}

// -----------------------------------------------------------------------
// MYLANG::set_module(char *mod_name,char **mod_list)
//
// Sets the module name.  Does nothing if it's already set (so it can
// be overridden as a command line option).
//
// mod_list is a NULL-terminated list of additional modules.  This
// is really only useful when building static executables.
//-----------------------------------------------------------------------
```

```
void MYLANG::set_module(char *mod_name, char **mod_list) {
  if (module) return;
  module = new char[strlen(mod_name)+1];
  strcpy(module,mod_name);
}

// ---------------------------------------------------------------------
// MYLANG::headers(void)
//
// Generate the appropriate header files for MYLANG interface.
// ---------------------------------------------------------------------
void MYLANG::headers(void) {
       emit_banner(f_header);       // Print the SWIG banner message
       fprintf(f_header,"/* Implementation : My Language */\n\n");
}

// ---------------------------------------------------------------------
// MYLANG::initialize(void)
//
// Produces an initialization function.   Assumes that the module
// name has already been specified.
// ---------------------------------------------------------------------
void MYLANG::initialize() {
       if (!module) module = “swig”;         // Pick a default name
       // Start generating the initialization function
       fprintf(f_init,"int %s_initialize() {\n", module);
}

// ---------------------------------------------------------------------
// MYLANG::close(void)
//
// Finish the initialization function. Close any additional files and
// resources in use.
// ---------------------------------------------------------------------
void MYLANG::close(void) {
       // Finish off our init function
       fprintf(f_init,"}\n");
}

// ---------------------------------------------------------------------
// MYLANG::create_command(char *cname, char *iname)
//
// Creates a new command from a C function.
//            cname = Name of the C function
//            iname = Name of function in scripting language
// ---------------------------------------------------------------------
void MYLANG::create_command(char *cname, char *iname) {
       fprintf(f_init,”\t Creating command %s\n”, iname);
}

// ---------------------------------------------------------------------
// MYLANG::create_function(char *name, char *iname, DataType *d, ParmList *l)
//
// Create a function declaration and register it with the interpreter.
//            name = Name of real C function
//            iname = Name of function in scripting language
//            d = Return datatype
//            l = Function parameters
// ---------------------------------------------------------------------
```

```
void MYLANG::create_function(char *name, char *iname, DataType *d, ParmList *l) {
       fprintf(f_wrappers,"\nwrap_%s() { }\n\n", name);
       create_command(name,iname);
}


// ---------------------------------------------------------------------
// MYLANG::link_variable(char *name, char *iname, DataType *t)
//
// Create a link to a C variable.
//            name = Name of C variable
//            iname = Name of variable in scripting language
//            t = Datatype of the variable
// ---------------------------------------------------------------------
void MYLANG::link_variable(char *name, char *iname, DataType *t) {
       fprintf(f_init,"\t Linking variable : %s\n", iname);
}

// ---------------------------------------------------------------------
// MYLANG::declare_const(char *name, char *iname, DataType *type, char *value)
//
// Makes a constant.
//            name = Name of the constant
//            iname = Scripting language name of constant
//            type = Datatype of the constant
//            value = Constant value (as a string)
// ---------------------------------------------------------------------
void MYLANG::declare_const(char *name, char *iname, DataType *type, char *value) {
       fprintf(f_init,"\t Creating constant : %s = %s\n", name, value);
}
```

To compile our new language, we write a main program (as described previously) and do this :

```
% g++ main.cxx mylang.cxx -I/usr/local/include -L/usr/local/lib -lswig -o myswig
```

Now, try running this new version of SWIG on a few interface files to see what happens. The various `printf()` statements will show you where output appears and how it is structured. For example, if we run this module on the following interface file :

```
/* File : example.i */
%module example
%{
/* Put headers and other declarations here */
%}

// A function
extern double foo(double a, double b);

// A variable
extern int bar;

// A constant
#define SPAM 42
```

We get the following output :

```
/*
```

```
 * FILE : example_wrap.c
 *
 * This file was automatically generated by :
 * Simplified Wrapper and Interface Generator (SWIG)
 * Version 1.1  (Final)
 *
 * Portions Copyright (c) 1995-1997
 * The University of Utah and The Regents of the University of California.
 * Permission is granted to distribute this file in any manner provided
 * this notice remains intact.
 *
 * Do not make changes to this file--changes will be lost!
 *
 */


#define SWIGCODE
/* Implementation : My Language */


/* Put headers and other declarations here */
extern double foo(double ,double );
extern int  bar;

wrap_foo() { }

int example_initialize() {
         Creating command foo
         Linking variable : bar
         Creating constant : SPAM = 42
}
```

Looking at the language module and the output gives some idea of how things are structured. The first part of the file is a banner message printed by the `emit_banner()` function. The "extern" declarations are automatically supplied by the SWIG compiler when use an extern modifier. The wrapper functions appear after all of the headers and forward declarations. Finally, the initialization function is written.

It is important to note that this minimal module is enough to use virtually all aspects of SWIG. If we feed SWIG a C++ file, we will see our low-level module functions being called even though we have not explicitly defined any C++ handling (this is due to the layered approach of implementing C++ on top of C). For example, the following interface file

```
%module example
struct Vector {
        double x,y,z;
        Vector();
        ~Vector();
        double magnitude();
};
```

produces accessor functions and the following output :

```
/*
 * FILE : example_wrap.c
 *
```

```
 * This file was automatically generated by :
 * Simplified Wrapper and Interface Generator (SWIG)
 * Version 1.1  (Final)
 *
 * Portions Copyright (c) 1995-1997
 * The University of Utah and The Regents of the University of California.
 * Permission is granted to distribute this file in any manner provided
 * this notice remains intact.
 *
 * Do not make changes to this file--changes will be lost!
 *
 */


#define SWIGCODE
/* Implementation : My Language */

static double  Vector_x_set(Vector *obj, double  val) {
    obj->x = val;
    return val;
}

wrap_Vector_x_set() { }

static double  Vector_x_get(Vector *obj) {
    double  result;
    result = (double ) obj->x;
    return result;
}

wrap_Vector_x_get() { }

static double  Vector_y_set(Vector *obj, double  val) {
    obj->y = val;
    return val;
}

wrap_Vector_y_set() { }

static double  Vector_y_get(Vector *obj) {
    double  result;
    result = (double ) obj->y;
    return result;
}

wrap_Vector_y_get() { }

static double  Vector_z_set(Vector *obj, double  val) {
    obj->z = val;
    return val;
}

wrap_Vector_z_set() { }

static double  Vector_z_get(Vector *obj) {
    double  result;
    result = (double ) obj->z;
    return result;
}
```

```
wrap_Vector_z_get() { }

static Vector *new_Vector() {
    return new Vector();
}

wrap_new_Vector() { }

static void delete_Vector(Vector *obj) {
    delete obj;
}

wrap_delete_Vector() { }

static double  Vector_magnitude(Vector *obj) {
    double  _result = (double )obj->magnitude();
    return _result;
}

wrap_Vector_magnitude() { }

int example_initialize() {
         Creating command Vector_x_set
         Creating command Vector_x_get
         Creating command Vector_y_set
         Creating command Vector_y_get
         Creating command Vector_z_set
         Creating command Vector_z_get
         Creating command new_Vector
         Creating command delete_Vector
         Creating command Vector_magnitude
}
```

With just a little work, we already see that SWIG does quite alot for us. Now our task is to fill in the various Language methods with the real code needed to produce a working module. Before doing that, we first need to take a tour of some important SWIG datatypes and functions.

A tour of SWIG datatypes

While SWIG has become somewhat complicated over the last year, its internal operation is based on just a few fundamental datatypes. These types are described now although examples of using the various datatypes are shown later.

The DataType class

All C datatypes are represented by the following structure :

```
class DataType {
public:
       DataType();
       DataType(DataType *);
       ~DataType();
       int         type;               // SWIG Type code
       char        name[MAXNAME];      // Name of type
```

```
        char        is_pointer;          // Is this a pointer?
        char        implicit_ptr;        // Implicit ptr
        char        is_reference;        // A C++ reference type
        char        status;              // Is this datatype read-only?
        char        *qualifier;          // A qualifier string (ie. const).
        char        *arraystr;           // String containing array part
        int         id;                  // type identifier (unique for every type).
        // Output methods
        char       *print_type();        // Return string containing datatype
        char       *print_full();        // Return string with full datatype
        char       *print_cast();        // Return string for type casting
        char       *print_mangle();      // Return mangled version of type
        char       *print_mangle_default(); // Default mangling scheme
        char       *print_real();        // Print the real datatype
        char       *print_arraycast();   // Prints an array cast
        // Array query functions
        int        array_dimensions();   // Return number of array dimensions (if any)
        char       *get_dimension(int);  // Return string for a particular dimension
};
```

The fundamental C datatypes are given a unique numerical code which is stored in the `type` field. The current list of types is as follows :

C Datatype	**SWIG Type Code**
int	T_INT
short	T_SHORT
long	T_LONG
char	T_CHAR
float	T_FLOAT
double	T_DOUBLE
void	T_VOID
unsigned int	T_UINT
unsigned short	T_USHORT
unsigned long	T_ULONG
unsigned char	T_UCHAR
signed char	T_SCHAR
bool	T_BOOL
<user>	T_USER
error	T_ERROR

The `T_USER` type is used for all derived datatypes including structures and classes. The `T_ERROR` type indicates that a parse/type error has occurred and went undetected (as far as I know this doesn't happen).

The `name[]` field contains the actual name of the datatype as seen by the parser and is currently limited to a maximum of 96 bytes (more than enough for most applications). If a typedef has been used, the name field contains the actual name used, not the name of the primitive C datatype. Here are some examples :

C Datatype	**type**	**name[]**
double	T_DOUBLE	double
unsigned int	T_UINT	unsigned int
signed long	T_LONG	signed long
struct Vector	T_USER	struct Vector
Real	T_DOUBLE	Real

C qualifiers such as "`const`" or "`volatile`" are stored separately in the `qualifier` field. In order to produce usable wrapper code, SWIG often needs to strip the qualifiers. For example, trying to assign a passed function argument into a type of "`const int`" will irritate most compilers. Unfortunately, this kind of assignment is unavoidable when converting arguments between a scripting and C representation.

The `is_pointer` field indicates whether or not a particular datatype is a pointer. The value of `is_pointer` determines the level of indirection used. For example :

C Datatype	**type**	**is_pointer**
double *	T_DOUBLE	1
int ***	T_INT	3
char *	T_CHAR	1

The `implicit_ptr` field is an internally used parameter that is used to properly handle the use of pointers in typedef statements. However, for the curious, in indicates the level of indirection implicitly defined in a datatype. For example :

```
typedef char *String;
```

is represented by a datatype with the following parameters :

```
type          = T_CHAR;
name[]        = "String";
is_pointer    = 1;
implicit_ptr  = 1;
```

Normally, language modules do not worry about the `implicit_ptr` field.

C++ references are indicated by the `is_reference` field. By default, the parser converts references into pointers which makes them indistinguishable from other pointer datatypes. However, knowing that something is a reference effects some code generation procedures so this field can be checked to see if a datatype really is a C++ reference.

The `arraystr` field is used to hold the array dimensions of array datatypes. The dimensions are simply represented by a string. For example :

C Datatype	type	is_pointer	arraystr
double a[50]	T_DOUBLE	1	[50]
int b[20][30][50]	T_INT	1	[20][30][50]
char *[MAX]	T_CHAR	2	[MAX]

SWIG converts all arrays into pointers. Thus a "`double [50]`" is really just a special version of "`double *`". If a datatype is not declared as an array, the `arraystr` field contains the NULL pointer.

A collection of "output" methods are available for datatypes. The names of these methods are mainly "historical" in that they don't actually "print" anything, but now return character strings instead. Assuming that `t` is a datatype representing the C datatype "`const int *`", here's what the methods produce :

Operation	Output
t->print_type()	int *
t->print_full()	const int *
t->print_cast()	(int *)
t->print_mangle()	< language dependent >
t->print_mangle_default()	_int_p

A few additional output methods are provided for dealing with arrays :

type	Operation	Output
int a[50]	t->print_type()	int *
int a[50]	t->print_real()	int [50]
int a[50]	t->print_arraycast()	(int *)
int a[50][50]	t->print_arraycast()	(int (*)[50])

Additional information about arrays is also available using the following functions :

type	Operation	Result
int a[50]	t->array_dimension()	1
int a[50]	t->get_dimension(0)	50
int b[MAXN][10]	t->array_dimension()	2
int b[MAXN][10]	t->get_dimension(0)	MAXN
int b[MAXN][10]	t->get_dimension(1)	10

The DataType class contains a variety of other methods for managing typedefs, scoping, and other operations. These are usually only used by the SWIG parser. While available to language modules too, they are never used (at least not in the current implementation), and should probably be avoided unless you absolutely know what you're doing (not that this is a strict requirement of course).

Function Parameters

Each argument of a function call is represented using the Parm structure :

```
struct Parm {
       Parm(DataType *type, char *name);
       Parm(Parm *p);
       ~Parm();
       DataType   *t;               // Datatype of this parameter
       int        call_type;        // Call type (value or reference or value)
       char       *name;            // Name of parameter (optional)
       char       *defvalue;        // Default value (as a string)
       int        ignore;           // Ignore flag
};
```

t is the datatype of the parameter, name is an optional parameter name, and defvalue is a default argument value (if supplied).

call_type is an integer code describing any special processing. It can be one of two values :

- CALL_VALUE. This means that the argument is a pointer, but we should make it work like a call-by-value argument in the scripting interface. This value used to be set by the %val directive, but this approach is now deprecated (since the same effect can be achieved by typemaps).

- `CALL_REFERENCE`. This is set when a complex datatype is being passed by value to a function. Since SWIG can't handle complex datatypes by value, the datatype is implicitly changed into a pointer and `call_type` set to `CALL_REFERENCE`. Many of SWIG's internals look at this when generating code. A common mistake is forgetting that all complex datatypes in SWIG are pointers. This is even the case when writing a language-module---the conversion to pointers takes place in the parser before data is even passed into a particular module.

The `ignore` field is set when SWIG detects that a function parameter is to be "ignored" when generating wrapper functions. An "ignored" parameter is usually set to a default value and effectively disappears when a function call is made from a scripting language (that is, the function is called with fewer arguments than are specified in the interface file). The `ignore` field is normally only set when an "ignore" typemap has been used.

All of the function parameters are passed in the structure `ParmList`. This structure has the following user-accesible methods available :

```
class ParmList {
public:
  int   nparms;                  // Number of parms in list
  void  append(Parm *p);         // Append a parameter to the end
  void  insert(Parm *p, int pos); // Insert a parameter into the list
  void  del(int pos);            // Delete a parameter at position pos
  int   numopt();                // Get number of optional arguments
  int   numarg();                // Get number of active arguments
  Parm *get(int pos);            // Get the parameter at position pos
};
```

The methods operate in the manner that you would expect. The most common operation that will be performed in a language module is walking down the parameter list and processing individual parameters. This can be done as follows :

```
// Walk down a parameter list
ParmList *l;                // Function parameter list (already created)
Parm *p;

for (int i = 0; i < l->nparms; i++) {
       p = l->get(i);              // Get ith parameter
       // do something with the parameter
       ...
}
```

The String Class

The process of writing wrapper functions is mainly just a tedious exercise in string manipulation. To make this easier, the `String` class provides relatively simple mechanism for constructing strings, concatenating values, replacing symbols, and so on.

```
class String {
public:
  String();
  String(const char *s);
  ~String();
```

```
  char  *get() const;
  friend String& operator<<(String&,const char *s);
  friend String& operator<<(String&,const int);
  friend String& operator<<(String&,const char);
  friend String& operator<<(String&,String&);
  friend String& operator>>(const char *s, String&);
  friend String& operator>>(String&,String&);
  String& operator=(const char *);
  operator char*() const { return str; }
  void   untabify();
  void   replace(char *token, char *rep);
  void   replaceid(char *id, char *rep);
};
```

Strings can be manipulated in a manner that looks similar to C++ I/O operations. For example :

```
String s;

s << "void" << " foo() {\n"
  << tab4 << "printf(\"Hello World\");\n"
  << "}\n";

fprintf(f_wrappers,"%s", (char *) s);
```

produces the output :

```
void foo() {
    printf("Hello World");
}
```

The `<<` operator always appends to the end of a string while `>>` can be used to insert a string at the beginning. Strings may be used anywhere a `char *` is expected. For example :

```
String s1,s2;
...
if (strcmp(s1,s2) == 0) {
       printf("Equal!\n");
}
```

The `get()` method can be used to explicitly return the `char *` containing the string data. The `untabify()` method replaces tabs with whitespace. The `replace()` method can be used to perform substring replacement and is used by typemaps. For example :

```
s.replace("$target","_arg3");
```

The `replaceid()` method can be used to replace valid C identifiers with a new value. C identifiers must be surrounded by white-space or other non-identifier characters. The `replace()` method does not have this restriction.

Hash Tables

Hash tables can be created using the `Hash` class :

```
class Hash {
public:
  Hash();
```

```
  ~Hash();
  int    add(const char *key, void *object);
  int    add(const char *key, void *object, void (*del)(void *));
  void  *lookup(const char *key);
  void   remove(const char *key);
  void  *first();
  void  *next();
  char  *firstkey();
  char  *nextkey();
};
```

Hash tables store arbitrary objects (cast to `void *`) with string keys. An optional object deletion function may be registered with each entry to delete objects when the hash is destroyed. Hash tables are primarily used for managing internal symbol tables although language modules may also use them to keep track of special symbols and other state.

The following hash table shows how one might keep track of real and renamed function names.

```
Hash wrapped_functions;

int add_function(char *name, char *renamed) {
       char *nn = new char[strlen(renamed)+1];
       strcpy(nn,renamed);
       if (wrapped_functions.add(name,nn) == -1) {
              printf("Function multiply defined!\n");
              delete [] nn;
              return -1;
       }
}
char *get_renamed(char *name) {
       char *rn = (char *) wrapped_functions.lookup(name);
       return rn;
}
```

The `remove()` method removes a hash-table entry. The `first()` and `next()` methods are iterators for extracting all of the hashed objects. They return NULL when no more objects are found in the hash. The `firstkey()` and `nextkey()` methods are iterators that return the hash keys. NULL is returned when no more keys are found.

The WrapperFunction class

Finally, a `WrapperFunction` class is available for simplifying the creation of wrapper functions. The class is primarily designed to organize code generation and provide a few supporting services. The class is defined as follows :

```
class WrapperFunction {
public:
  String  def;
  String  locals;
  String  code;
  void    print(FILE *f);
  void    print(String &f);
  void    add_local(char *type, char *name, char *defvalue = 0);
  char   *new_local(char *type, char *name, char *defvalue = 0);
};
```

Three strings are available. The `def` string contains the actual function declaration, the `locals` string contain local variable declarations, and the `code` string contains the resulting wrapper code.

The method `add_local()` creates a new local variable which is managed with an internal symbol table (that detects variable conflicts and reports potential errors). The `new_local()` method can be used to create a new local variable that is guaranteed to be unique. Since a renaming might be required, this latter method returns the name of the variable that was actually selected for use (typically, this is derived from the original name).

The `print()` method can be used to emit the wrapper function to a file. The printing process consolidates all of the strings into a single result.

Here is a very simple example of the wrapper function class :

```
WrapperFunction f;

f.def << "void count_n(int n) {";
f.add_local("int","i");
f.code << tab4 << "for (i = 0; i < n; i++) {\n"
       << tab8 << "printf(\"%d\\n\",i);\n"
       << tab4 << "}\n"
       << "}\n";

f.print(f_wrappers);
```

This produces the following output :

```
void count_n(int n) {
    int i;
    for (i = 0; i < n; i++) {
         printf("%d\n",i);
    }
}
```

Of course, as you can guess, the functions actually generated by your language module will be more complicated than this.

Typemaps (from C)

The typemapper plays a big role in generating code for all of SWIG's modules. Understanding the `%typemap` directive and how it works is probably a good starting point for understanding this section.

The typemap C API.

There is a relatively simple C API for managing typemaps in language modules.

```
void typemap_register(char *op, char *lang, DataType *type, char *pname,
                      char *code, ParmList *l = 0);
```

Registers a new typemap with the typemapper. This is the C equivalent of the `%typemap` directive. For example :

```
%typemap(lang,op) type pname { ... code ... };
%typemap(lang,op) type pname(ParmList) { ... code ... };
```

`code` contains the actual typemap code, while `l` is a parameter list containing local variable declarations. Normally, it is not necessary to execute this function from language modules.

```
void typemap_register_default(char *op, char *lang, int type, int ptr,
                              char *arraystr,  char *code, ParmList *args)
```

Registers a default typemap. This works in the same way as the normal registration function, but takes a type code (an integer) instead. Default typemaps are more general than normal typemaps (see below).

```
char *typemap_lookup(char *op, char *lang, DataType *type,
                     char *pname, char *source, char *target,
                     WrapperFunction *f = 0);
```

Looks up a typemap, performs variable substitutions, and returns a string with the corresponding typemap code. The tuple `(op, lang, type, pname)` determine which typemap to find. `source` contains the string to be assigned to the `$source` variable, `target` contains the string to be assigned to the `$target` variable. `f` is an optional wrapper function object. It should be supplied to support parameterized typemaps (ie. typemaps that declare additional local variables).

`typemap_lookup()` returns NULL is no typemap is found. Otherwise it returns a string containing the updated typemap code. This code has a number of variables substituted including `$source`, `$target`, `$type`, `$mangle`, and `$basetype`.

```
char *typemap_check(char *op, char *lang, DataType *type, char *pname)
```

Checks to see if a typemap exists. The tuple `(op, lang, type, pname)` determine which typemap to find. If the typemap exists, the raw typemap code is returned. Otherwise, NULL is returned. While `typemap_lookup()` could be used to accomplish the same thing, this function is more compact and is significantly faster since it does not perform any variable substitutions.

What happens on typemap lookup?

When looking for a typemap, SWIG searches for a match in a series of steps.

- Explicit typemaps. These are specified directly with the `%typemap()` directive. Named typemaps have the highest precedence while arrays have higher precedence than pointers (see the typemap chapter for more details).
- If no explicit typemap is found, mappings applied with the `%apply` directive are checked. Basically, `%apply` is nothing more than a glorified renaming operation. We rename the datatype and see if there are any explicit typemaps that match. If so, we use the typemap that was found.
- Default typemaps. If no match is found with either explicit typemaps or apply direc-

tives, we make a final search using the default typemap. Unlike other typemaps, default typemaps are applied to the raw SWIG internal datatypes (`T_INT`, `T_DOUBLE`, `T_CHAR`, etc...). As a result, they are insentive to typedefs and renaming operations. If nothing is found here, a NULL pointer is returned indicating that no mapping was found for that particular datatype.

Another way to think of the typemap mechanism is that it always tries to apply the most specific typemap that can be found for any particular datatype. When searching, it starts with the most specific and works its way out to the most general specification. If nothing is found it gives up and returns a NULL pointer.

How many typemaps are there?

All typemaps are identified by an operation string such as “in”, “out”, “memberin”, etc... A number of typemaps are defined by other parts of SWIG, but you can create any sort of typemap that you wish by simply picking a new name and using it when making calls to `typemap_lookup()` and `typemap_check()`.

File management

The following functions are provided for managing files within SWIG.

`void add_directory(char *dirname);`

Adds a new directory to the search path used to locate SWIG library files. This is the C equivalent of the `swig -I` option.

`int insert_file(char *filename, FILE *output);`

Searches for a file and copies it into the given output stream. The search process goes through the SWIG library mechanism which first checks the current directory, then in various parts of the SWIG library for a match. Returns -1 if the file is not found. Language modules often use this function to insert supporting code. Usually these code fragments are given a ‘`.swg`’ suffix and are placed in the SWIG library.

`int get_file(char *filename, String &str);`

Searches for a file and returns its contents in the String `str`. Returns a -1 if the file is not found.

`int checkout_file(char *source, char *dest);`

Copies a file from the SWIG library into the current directory. `dest` is the filename of the desired file. This function will not replace a file that already exists. The primary use of this function is to give the user supporting code. For example, we could check out a Makefile if none exists. Returns -1 on failure.

`int include_file(char *filename);`

The C equivalent of the SWIG `%include` directive. When called, SWIG will attempt to

open `filename` and start parsing all of its contents. If successful, parsing of the new file will take place immediately. When the end of the file is reached, the parser switches back to the input file being read prior to this call. Returns -1 if the file is not found.

Naming Services

The naming module provides methods for generating the names of wrapper functions, accessor functions, and other aspects of SWIG. Each function returns a new name that is a syntactically correct C identifier where invalid characters have been converted to a "_".

`char *name_wrapper(char *fname, char *prefix);`
Returns the name of a wrapper function. By default, it will be "`_wrap_prefixfname`".

`char *name_member(char *mname, char *classname;`
Returns the name of a C++ accessor function. Normally, this is "`classname_mname`".

`char *name_get(char *vname);`
Returns the name of a function to get the value of a variable or class data member. Normally "`vname_get`" is returned.

`char *name_set(char *vname);`
Returns the name of a function to set the value of a variable or class data member. Normally, "`vname_set`" is returned.

`char *name_construct(char *classname);`
Returns the name of a constructor function. Normally returns "`new_classname`".

`char *name_destroy(char *classname);`
Returns the name of a destructor function. Normally returns "`delete_classname`".

Each function may also accept an optional parameter of `AS_IS`. This suppresses the conversion of illegal characters (a process that is sometimes required). For example :

```
char *name = name_member("foo","bar",AS_IS);   // Produce a name, but don't change
                                               // illegal characters.
```

It is critical that language modules use the naming functions. These function are used throughout SWIG and provide a centralized mechanism for keeping track of functions that have been generated, managing multiple files, and so forth. In future releases, it may be possible to change the naming scheme used by SWIG. Using these functions should insure future compatibility.

Code Generation Functions

The following functions are used to emit code that is generally useful and used in essentially every SWIG language module.

`int emit_args(DataType *t, ParmList *l, WrapperFunction &f);`
Creates all of the local variables used for function arguments and return value. `t` is the return datatype of the function, `l` is the parameter list holding all of the function argu-

ments. `f` is a WrapperFunction object where the local variables will be created.

`void emit_func_call(char *name, DataType *t, ParmList *l, WrapperFunction &f);`

Creates a function call to a C function. `name` is the name of the C function, `t` is the return datatype, `l` is the function parameters and `f` is a WrapperFunction object. The code generated by this function assumes that one has first called `emit_args()`.

`void emit_banner(FILE *file);`

Emits the SWIG banner comment to the output file.

`void emit_set_get(char *name, char *rename, DataType *t);`

Given a variable of type `t`, this function creates two functions to set and get the value. These functions are then wrapped like normal C functions. `name` is the real name of the variable. `rename` is the renamed version of the variable.

`void emit_ptr_equivalence(FILE *file);`

To keep track of datatypes, SWIG maintains an internal table of "equivalent" datatypes. This table is updated by typedef, class definitions, and other C constructs. This function emits code compatible with the type-checker that is needed to make sure pointers work correctly. Typically this function is called after an interface file has been parsed completely.

Writing a Real Language Module

Whew, assuming you've made it this far, we're ready to write a real language module. In this example, we'll develop a simple Tcl module. Tcl has been chosen because it has a relatively simple C API that is well documented and easy to understand. The module developed here is not the same as the real SWIG Tcl module (which is significantly more complicated).

The header file

We will start with the same header file as before :

```
// File : mylang.h
// A simple SWIG Language module

class MYLANG : public Language {
private:
       char *module;
public :
       MYLANG() {
              module = 0;
       };
       // Virtual functions required by the SWIG parser
        void parse_args(int, char *argv[]);
        void parse();
        void create_function(char *, char *, DataType *, ParmList *);
        void link_variable(char *, char *, DataType *);
        void declare_const(char *, char *, DataType *, char *);
        void initialize(void);
        void headers(void);
        void close(void);
```

```
        void set_module(char *,char **);
        void create_command(char *, char *);
};
```

Command Line Options and Basic Initialization

Command line options are parsed using the `parse_args()` method :

```
// -----------------------------------------------------------------------
// A simple SWIG Language module
//
// -----------------------------------------------------------------------

#include "swig.h"
#include "mylang.h"

static char *usage = "\
My Language Options\n\
     -module name    - Set name of module\n\n";

// ---------------------------------------------------------------------
// MYLANG::parse_args(int argc, char *argv[])
//
// Parse my command line options and initialize by variables.
// ---------------------------------------------------------------------

void MYLANG::parse_args(int argc, char *argv[]) {
  // Look for certain command line options
  for (int i = 1; i < argc; i++) {
    if (argv[i]) {
      if (strcmp(argv[i],"-module") == 0) {
        if (argv[i+1]) {
          set_module(argv[i+1],0);
          mark_arg(i);
          mark_arg(i+1);
          i++;
        } else {
          arg_error();
        }
      } else if (strcmp(argv[i],"-help") == 0) {
        fprintf(stderr,"%s\n", usage);
      }
    }
  }
  // Set location of SWIG library
  strcpy(LibDir,"tcl");

  // Add a symbol to the parser for conditional compilation
  add_symbol("SWIGTCL",0,0);

  // Add typemap definitions
  typemap_lang = "tcl";
}
```

Parsing command line options follows the same conventions as for writing a C++ main program with one caveat. For each option that your language module parses, you need to call the function `mark_arg()`. This tells the SWIG main program that your module found a valid option

and used it. If you don't do this, SWIG will exit with an error message about unrecognized command line options.

After processing command line options, you next need to set the variable `LibDir` with the name of the subdirectory your language will use to find files in the SWIG library. Since we are making a new Tcl module, we'll just set this to "`tcl`".

Next, we may want to add a symbol to SWIG's symbol table. In this case we're adding "`SWIGTCL`" to indicate that we're using Tcl. SWIG modules can use this for conditional compilation and detecting your module using `#ifdef`.

Finally, we need to set the variable `typemap_lang`. This should be assigned a name that you would like to use for all typemap declarations. When a user gives a typemap, they would use this name as the target language.

Starting the parser

To start the SWIG parser, the `parse()` method is used :

```
// ----------------------------------------------------------------------
// void MYLANG::parse()
//
// Start parsing an interface file for MYLANG.
// ----------------------------------------------------------------------

void MYLANG::parse() {

       fprintf(stderr,"Making wrappers for Tcl\n");
       headers();       // Emit header files and other supporting code

       // Tell the parser to first include a typemap definition file

       if (include_file("lang.map") == -1) {
           fprintf(stderr,"Unable to find lang.map!\n");
           SWIG_exit(1);
       }
       yyparse();       // Run the SWIG parser
}
```

This function should print some kind of message to the user indicating what language is being targeted. The `headers()` method is called (see below) to emit support code and header files. Finally, we make a call to `yyparse()`. This starts the SWIG parser and does not return until the entire interface file has been read.

In our implementation, we have also added code to immediately include a file '`lang.map`'. This file will contain typemap definitions to be used by our module and is described in detail later.

Emitting headers and support code

Prior to emitting any code, our module should emit standard header files and support code. This is done using the `headers()` method :

```
// ----------------------------------------------------------------------
// MYLANG::headers(void)
//
```

```
// Generate the appropriate header files for MYLANG interface.
// ----------------------------------------------------------------------

void MYLANG::headers(void)
{
  emit_banner(f_header);              // Print the SWIG banner message
  fprintf(f_header,"/* Implementation : My TCL */\n\n");

  // Include header file code fragment into the output
  if (insert_file("header.swg",f_header) == -1) {
    fprintf(stderr,"Fatal Error. Unable to locate 'header.swg'.\n");
    SWIG_exit(1);
  }

  // Emit the default SWIG pointer type-checker (for strings)
  if (insert_file("swigptr.swg",f_header) == -1) {
    fprintf(stderr,"Fatal Error. Unable to locate 'swigptr.swg'.\n");
    SWIG_exit(1);
  }
}
```

In this implementation, we emit the standard SWIG banner followed by a comment indicating which language module is being used. After that, we are going to include two different files. '`header.swg`' is a file containing standard declarations needed to build a Tcl extension. For our example, it will look like this :

```
/* File : header.swg */
#include <tcl.h>
```

The file '`swigptr.swg`' contains the standard SWIG pointer-type checking library. This library contains about 300 lines of rather nasty looking support code that define the following 3 functions :

`void SWIG_RegisterMapping(char *type1, char *type, void *(*cast)(void *))`

Creates a mapping between C datatypes `type1` and `type2`. This is registered with the runtime type-checker and is similiar to a typedef. `cast` is an optional function pointer defining a method for proper pointer conversion (if needed). Normally, cast is only used when converting between base and derived classes in C++ and is needed for proper implementation of multiple inheritance.

`void SWIG_MakePtr(char *str, void *ptr, char *type);`

Makes a string representation of a C pointer. The result is stored in `str` which is assumed to be large enough to hold the result. `ptr` contains the pointer value and `type` is a string code corresponding to the datatype.

`char *SWIG_GetPtr(char *str, void **ptr, char *type);`

Extracts a pointer from its string representation, performs type-checking, and casting. `str` is the string containing the pointer-value representation, `ptr` is the address of the pointer that will be returned, and `type` is the string code corresponding to the datatype. If a type-error occurs, the function returns a `char *` corresponding to the part of the input string that was invalid, otherwise the function returns NULL. If a NULL pointer is given for `type`, the function will accept a pointer of any type.

We will use these functions later.

Setting a module name

The `set_module()` method is used whenever the `%module` directive is encountered.

```
// ----------------------------------------------------------------------
// MYLANG::set_module(char *mod_name,char **mod_list)
//
// Sets the module name.  Does nothing if it's already set (so it can
// be overriddent as a command line option).
//
// mod_list is a NULL-terminated list of additional modules to initialize
// and is only provided if the user specifies something like this :
// %module foo, mod1, mod2, mod3, mod4
//----------------------------------------------------------------------

void MYLANG::set_module(char *mod_name, char **mod_list) {
       if (module) return;
       module = new char[strlen(mod_name)+1];
       strcpy(module,mod_name);
       // Make sure the name conforms to Tcl naming conventions
       for (char *c = module; (*c); c++)
              *c = tolower(*c);
       toupper(module);
}
```

This function may, in fact, be called multiple times in the course of processing. Normally, we only allow a module name to be set once and ignore all subsequent calls however.

Final Initialization

The initialization of a module takes several steps--parsing command line options, printing standard header files, starting the parser, and setting the module name. The final step in initialization is calling the `initialize()` method :

```
// ----------------------------------------------------------------------
// MYLANG::initialize(void)
//
// Produces an initialization function.   Assumes that the module
// name has already been specified.
// ----------------------------------------------------------------------

void MYLANG::initialize()
{
       // Check if a module has been defined
       if (!module) {
              fprintf(stderr,"Warning. No module name given!\n");
              module = "swig";
       }

       // Generate a CPP symbol containing the name of the initialization function
       fprintf(f_header,"#define SWIG_init    %s_Init\n\n\n", module);

       // Start generating the initialization function
       fprintf(f_init,"int SWIG_init(Tcl_Interp *interp) {\n");
```

```
        fprintf(f_init,"\t if (interp == 0) return TCL_ERROR;\n");
}
```

The `initialize()` method should create the module initialization function by emitting code to `f_init` as shown. By this point, we should already know the name of the module, but should check just in case. The preferred style of creating the initialization function is to create a C preprocessor symbol `SWIG_init`. Doing so may look weird, but it turns out that many SWIG library files may want to know the name of the initialization function. If we define a symbol for it, these files can simply assume that it's called `SWIG_init()` and everything will work out (okay, so it's a hack).

Cleanup

When an interface file has finished parsing, we need to clean everything up. This is done using the `close()` method :

```
// ---------------------------------------------------------------------
// MYLANG::close(void)
//
// Wrap things up.  Close initialization function.
// ---------------------------------------------------------------------

void MYLANG::close(void)
{
  // Dump the pointer equivalency table
  emit_ptr_equivalence(f_init);

  // Finish off our init function and print it to the init file
  fprintf(f_init,"\t return TCL_OK;\n");
  fprintf(f_init,"}\n");
}
```

The `close()` method should first call `emit_ptr_equivalence()` if the SWIG pointer type checker has been used. This dumps out support code to make sure the type-checker works correctly. Afterwards, we simply need to terminate our initialization function as shown. After this function has been called, SWIG dumps out all of its documentation files and exits.

Creating Commands

Now, we're moving into the code generation part of SWIG. The first step is to make a function to create scripting language commands. This is done using the `create_command()` function :

```
// ---------------------------------------------------------------------
// MYLANG::create_command(char *cname, char *iname)
//
// Creates a Tcl command from a C function.
// ---------------------------------------------------------------------
void MYLANG::create_command(char *cname, char *iname) {
        // Create a name for the wrapper function
        char *wname = name_wrapper(cname,"");

        // Create a Tcl command
        fprintf(f_init,"\t Tcl_CreateCommand(interp,\"%s\",%s, (ClientData) NULL,
                (Tcl_CmdDeleteProc *) NULL);\n", iname, wname);
}
```

For our Tcl module, this just calls Tcl_CreateCommand to make a new scripting language command.

Creating a Wrapper Function

The most complicated part of writing a language module is the process of creating wrapper functions. This is done using the `create_function()` method as shown here :

```
// ----------------------------------------------------------------------
// MYLANG::create_function(char *name, char *iname, DataType *d, ParmList *l)
//
// Create a function declaration and register it with the interpreter.
// ----------------------------------------------------------------------

void MYLANG::create_function(char *name, char *iname, DataType *t, ParmList *l)
{
  String           source, target;
  char             *tm;
  String           cleanup, outarg;
  WrapperFunction  f;

  // Make a wrapper name for this function
  char *wname = name_wrapper(iname,"");

  // Now write the wrapper function itself
  f.def << "static int " << wname << "(ClientData clientData, Tcl_Interp *interp, int
argc, char *argv[]) {\n";

  // Emit all of the local variables for holding arguments.
  int pcount = emit_args(t,l,f);

  // Get number of optional/default arguments
  int numopt = l->numopt();

  // Emit count to check the number of arguments
  f.code << tab4 << "if ((argc < " << (pcount-numopt) + 1 << ") || (argc > "
         << l->numarg()+1 << ")) {\n"
         << tab8 << "Tcl_SetResult(interp, \"Wrong # args.\",TCL_STATIC);\n"
         << tab8 << "return TCL_ERROR;\n"
         << tab4 << "}\n";

  // Now walk the function parameter list and generate code to get arguments
  int j = 0;               // Total number of non-optional arguments

  for (int i = 0; i < pcount ; i++) {
    Parm &p = (*l)[i];         // Get the ith argument
    source = "";
    target = "";

    // Produce string representation of source and target arguments
    source << "argv[" << j+1 << "]";
    target << "_arg" << i;
    if (!p.ignore) {
      if (j >= (pcount-numopt))  // Check if parsing an optional argument
        f.code << tab4 << "if argc >" << j+1 << ") {\n";

      // Get typemap for this argument
      tm = typemap_lookup("in",typemap_lang,p.t,p.name,source,target,&f);
```

```
    if (tm) {
      f.code << tm << "\n";
      f.code.replace("$arg",source);   // Perform a variable replacement
    } else {
      fprintf(stderr,"%s : Line %d. No typemapping for datatype %s\n",
              input_file,line_number, p.t->print_type());
    }
    if (j >= (pcount-numopt))
      f.code << tab4 << "} \n";
    j++;
  }

  // Check to see if there was any sort of a constaint typemap
  if ((tm = typemap_lookup("check",typemap_lang,p.t,p.name,source,target))) {
    f.code << tm << "\n";
    f.code.replace("$arg",source);
  }

  // Check if there was any cleanup code (save it for later)
  if ((tm = typemap_lookup("freearg",typemap_lang,p.t,p.name,target,
                           "interp->result"))) {
    cleanup << tm << "\n";
    cleanup.replace("$arg",source);
  }
  if ((tm = typemap_lookup("argout",typemap_lang,p.t,p.name,target,
                           "interp->result"))) {
    outarg << tm << "\n";
    outarg.replace("$arg",source);
  }
}

// Now write code to make the function call
emit_func_call(name,t,l,f);

// Return value if necessary
if ((t->type != T_VOID) || (t->is_pointer)) {
  if ((tm = typemap_lookup("out",typemap_lang,t,name,"_result","interp->result"))) {
    // Yep.  Use it instead of the default
    f.code << tm << "\n";
  } else {
    fprintf(stderr,"%s : Line %d. No return typemap for datatype %s\n",
            input_file,line_number,t->print_type());
  }
}

// Dump argument output code;
f.code << outarg;

// Dump the argument cleanup code
f.code << cleanup;

// Look for any remaining cleanup.  This processes the %new directive
if (NewObject) {
  if ((tm = typemap_lookup("newfree",typemap_lang,t,iname,"_result",""))) {
    f.code << tm << "\n";
  }
}

// Special processing on return value.
```

```
  if ((tm = typemap_lookup("ret",typemap_lang,t,name,"_result",""))) {
    f.code << tm << "\n";
  }

  // Wrap things up (in a manner of speaking)
  f.code << tab4 << "return TCL_OK;\n}";

  // Substitute the cleanup code (some exception handlers like to have this)
  f.code.replace("$cleanup",cleanup);

  // Emit the function
  f.print(f_wrappers);

  // Now register the function with the language
  create_command(iname,iname);
}
```

Creating a wrapper function really boils down to 3 components :

- Emit local variables and handling input arguments.
- Call the real C function.
- Convert the return value to a scripting language representation.

In our implementation, most of this work is done using typemaps. In fact, the role of the C++ code is really just to process typemaps in the appropriate order and to combine strings in the correct manner. The following typemaps are used in this procedure :

- "in". This is used to convert function arguments from Tcl to C.
- "out". This is used to convert the return value from C to Tcl.
- "check". This is used to apply constraints to the input values.
- "argout". Used to return values through function parameters.
- "freearg". Used to clean up arguments after a function call (possibly to release memory, etc...)
- "ret". Used to clean up the return value of a C function (possibly to release memory).
- "newfree" this is special processing applied when the `%new` directive has been used. Usually its used to clean up memory.

It may take awhile for this function to sink in, but its operation will hopefully become more clear shortly.

Manipulating Global Variables

To provide access to C global variables, the `link_variable()` method is used. In the case of Tcl, only `int`, `double`, and `char *` datatypes can be safely linked.

```
// ---------------------------------------------------------------------
// MYLANG::link_variable(char *name, char *iname, DataType *t)
//
// Create a Tcl link to a C variable.
// ---------------------------------------------------------------------

void MYLANG::link_variable(char *name, char *iname, DataType *t) {
  char *tm;
```

```
  // Uses a typemap to stick code into the module initialization function
  if ((tm = typemap_lookup("varinit",typemap_lang,t,name,name,iname))) {
    String temp = tm;
    if (Status & STAT_READONLY)
      temp.replace("$status"," | TCL_LINK_READ_ONLY");
    else
      temp.replace("$status","");
    fprintf(f_init,"%s\n",(char *) temp);
  } else {
    fprintf(stderr,"%s : Line %d. Unable to link with variable type %s\n",
            input_file,line_number,t->print_type());
  }
}
```

In this case, the procedure is looking for a typemap "varinit". We'll use the code specified with this typemap to create variable links. If no typemap is supplied or the user gives an unsupported datatypes, a warning message will be generated.

It is also worth noting that the `Status` variable contains information about whether or not a variable is read-only or not. To test for this, use the technique shown in the code above. Read-only variables may require special processing as shown.

Constants

Finally, creating constants is accomplished using the `declare_const()` method. For Tcl, we could do this :

```
// ----------------------------------------------------------------------
// MYLANG::declare_const(char *name, char *iname, DataType *type, char *value)
//
// Makes a constant.
// ----------------------------------------------------------------------

void MYLANG::declare_const(char *name, char *iname, DataType *type, char *value) {

  char *tm;
  if ((tm = typemap_lookup("const",typemap_lang,type,name,name,iname))) {
    String str = tm;
    str.replace("$value",value);
    fprintf(f_init,"%s\n", (char *) str);
  } else {
    fprintf(stderr,"%s : Line %d. Unable to create constant %s = %s\n",
            input_file, line_number, type->print_type(), value);
  }
}
```

We take the same approach used to create variables. In this case, the '`const`' typemap specifies the special processing.

The value of a constant is a string produced by the SWIG parser. It may contain an arithmetic expression such as "3 + 4*(7+8)". Because of this, it is critical to use this string in a way that allows it to be evaluated by the C compiler (this will be apparent when the typemaps are given).

A Quick Intermission

We are now done writing all of the methods for our language class. Of all of the methods,

`create_function()` is the most complicated and tends to do most of the work. We have also ignored issues related to documentation processing and C++ handling (although C++ will work with the functions we have defined so far).

While our C++ implementation is done, we still do not have a working language module. In fact, if we run SWIG on the following interface file :

```
/* File : example.i */
%module example
%{
/* Put headers and other declarations here */
%}

// A function
extern double foo(double a, double b);

// A variable
extern int bar;

// A constant
#define SPAM 42
```

we get the following errors :

```
[beazley@guinness lang]$ ./myswig example.i
Making wrappers for My Tcl
example.i : Line 9. No typemapping for datatype double
example.i : Line 9. No typemapping for datatype double
example.i : Line 9. No return typemap for datatype double
example.i : Line 12. Unable to link with variable type int
example.i : Line 16. Unable to create constant int  = 42
[beazley@guinness lang]$
```

The reason for this is that we have not yet defined any processing for real datatypes. For example, our language module has no idea how to convert doubles into Tcl strings, how to link with C variables and so on. To do this, we need to write a collection of typemaps.

Writing the default typemaps

In our earlier `parse()` method, there is a statement to include the file '`lang.map`'. We will use this file to write typemaps for our new language module. The '`lang.map`' file will actually go through the SWIG parser so we can write our typemaps using the normal `%typemap` directive. This approach makes it easy for us to debug and test our module because the typemaps can be developed and tested without having to repeatedly recompile the C++ part of the module.

Without further delay, here is the typemap file for our module (you might want to sit down) :

```
// ----------------------------------------------------------------------
// lang.map
//
// This file defines all of the type-mappings for our language (TCL).
// A typemap of 'SWIG_DEFAULT_TYPE' should be used to create default
// mappings.
// ----------------------------------------------------------------------
```

```
/*************************** FUNCTION INPUTS ***************************/

// Integers
%typemap(in) int             SWIG_DEFAULT_TYPE,
             short           SWIG_DEFAULT_TYPE,
             long            SWIG_DEFAULT_TYPE,
             unsigned int    SWIG_DEFAULT_TYPE,
             unsigned short  SWIG_DEFAULT_TYPE,
             unsigned long   SWIG_DEFAULT_TYPE,
             signed char     SWIG_DEFAULT_TYPE,
             unsigned char   SWIG_DEFAULT_TYPE
{
  int  temp;
  if (Tcl_GetInt(interp, $source, &temp) == TCL_ERROR) return TCL_ERROR;
  $target = ($type) temp;
}

// Floating point
%typemap(in) float   SWIG_DEFAULT_TYPE,
             double  SWIG_DEFAULT_TYPE
{
  double temp;
  if (Tcl_GetDouble(interp, $source, &temp) == TCL_ERROR) return TCL_ERROR;
  $target = ($type) temp;
}

// Strings
%typemap(in) char *  SWIG_DEFAULT_TYPE
{
  $target = $source;
}

// void *
%typemap(in) void *  SWIG_DEFAULT_TYPE
{
  if (SWIG_GetPtr($source,(void **) &$target, (char *) 0)) {
    Tcl_SetResult(interp,"Type error.  Expected a pointer",TCL_STATIC);
    return TCL_ERROR;
  }
}

// User defined types and all other pointers
%typemap(in) User * SWIG_DEFAULT_TYPE
{
  if (SWIG_GetPtr($source,(void **) &$target, "$mangle")) {
    Tcl_SetResult(interp,"Type error.  Expected a $mangle",TCL_STATIC);
    return TCL_ERROR;
  }
}

/************************** FUNCTION OUTPUTS ***************************/

// Signed integers
%typemap(out) int         SWIG_DEFAULT_TYPE,
              short       SWIG_DEFAULT_TYPE,
              long        SWIG_DEFAULT_TYPE,
              signed char SWIG_DEFAULT_TYPE
{
  sprintf($target,"%ld", (long) $source);
```

```
}

// Unsigned integers
%typemap(out) unsigned       SWIG_DEFAULT_TYPE,
              unsigned short SWIG_DEFAULT_TYPE,
              unsigned long  SWIG_DEFAULT_TYPE,
              unsigned char  SWIG_DEFAULT_TYPE
{
  sprintf($target,"%lu", (unsigned long) $source);
}

// Floating point
%typemap(out) double SWIG_DEFAULT_TYPE,
              float  SWIG_DEFAULT_TYPE
{
  Tcl_PrintDouble(interp,(double) $source,interp->result);
}

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

// Pointers
%typemap(out) User *SWIG_DEFAULT_TYPE
{
  SWIG_MakePtr($target,(void *) $source, "$mangle");
}

/************************** VARIABLE CREATION **************************/

// Integers
%typemap(varinit) int          SWIG_DEFAULT_TYPE,
                  unsigned int SWIG_DEFAULT_TYPE
{
  Tcl_LinkVar(interp, "$target", (char *) &$source, TCL_LINK_INT $status);
}

// Doubles
%typemap(varinit) double SWIG_DEFAULT_TYPE {
  Tcl_LinkVar(interp,"$target", (char *) &$source, TCL_LINK_DOUBLE $status);
}

// Strings
%typemap(varinit) char * SWIG_DEFAULT_TYPE {
  Tcl_LinkVar(interp,"$target", (char *) &$source, TCL_LINK_STRING $status);
}

/***************************** CONSTANTS *******************************/

// Signed Integers
%typemap(const) int           SWIG_DEFAULT_TYPE,
                short         SWIG_DEFAULT_TYPE,
                long          SWIG_DEFAULT_TYPE,
                signed char   SWIG_DEFAULT_TYPE
{
  static char *_wrap_$target;
  _wrap_$target = (char *) malloc(40);
```

```
  sprintf(_wrap_$target,"%ld",$value);
  Tcl_LinkVar(interp,"$target", (char *) &_wrap_$target, TCL_LINK_STRING |
TCL_LINK_READ_ONLY);
}

// Unsigned integers
%typemap(const) unsigned        SWIG_DEFAULT_TYPE,
                unsigned short  SWIG_DEFAULT_TYPE,
                unsigned long   SWIG_DEFAULT_TYPE,
                unsigned char   SWIG_DEFAULT_TYPE
{
  static char *_wrap_$target;
  _wrap_$target = (char *) malloc(40);
  sprintf(_wrap_$target,"%lu",$value);
  Tcl_LinkVar(interp,"$target", (char *) &_wrap_$target, TCL_LINK_STRING |
TCL_LINK_READ_ONLY);
}

// Doubles and floats
%typemap(const) double SWIG_DEFAULT_TYPE,
                float  SWIG_DEFAULT_TYPE
{
  static char *_wrap_$target;
  _wrap_$target = (char *) malloc(40);
  sprintf(_wrap_$target,"%f",$value);
  Tcl_LinkVar(interp,"$target", (char *) &_wrap_$target, TCL_LINK_STRING |
TCL_LINK_READ_ONLY);
}

// Strings
%typemap(const) char *SWIG_DEFAULT_TYPE
{
  static char *_wrap_$target = "$value";
  Tcl_LinkVar(interp,"$target", (char *) &_wrap_$target, TCL_LINK_STRING |
TCL_LINK_READ_ONLY);
}

// Pointers
%typemap(const) User *SWIG_DEFAULT_TYPE
{
  static char *_wrap_$target;
  _wrap_$target = (char *) malloc(20+strlen("$mangle"));
  SWIG_MakePtr(_wrap_$target, (void *) $value, "$mangle");
  Tcl_LinkVar(interp,"$target", (char *) &_wrap_$target, TCL_LINK_STRING |
TCL_LINK_READ_ONLY);
}
```

Now that we have our typemaps file, we are done and can start producing a variety of interesting Tcl extension modules. Should errors arrise, one will either have to pry into the C++ module or the typemaps file for a correction.

The SWIG library and installation issues

To make a new SWIG module generally usable, you will want to perform the following steps :

- Put the new binary in a publicly accessible location (ie. `/usr/local/bin`).
- Make a subdirectory for your language in the SWIG library. The library should match up

with the name you assigned to the `LibDir` variable in `parse_args()`.
- Copy the file '`lang.map`' to the SWIG library directory. Your new version of SWIG will now be able to find it no matter what directory SWIG is executed from.
- Provide some documentation about how your module works.

SWIG extensions are only able to target a single scripting language. If you would like to make your module part of the full version of SWIG, you will need to modify the file '`swigmain.cxx`' in the `SWIG1.1/Modules` directory. To do this, follow these steps :

- Add a `#include "lang.h"` to the `swigmain.cxx` file.
- Create a command line option for your module and write code to create an instance of your language (just copy the technique used for the other languages).
- Modify `Modules/Makefile` to include your module as part of its compilation process.
- Rebuild SWIG by typing 'make'.

C++ Processing

Language modules have the option to provide special processing for C++ classes. Usually this is to provide some sort of object-oriented interface such as shadow classes. The process of developing these extensions is highly technical and the best approach may be to copy pieces from other SWIG modules that provide object oriented support.

How C++ processing works

The wrapping of C++ classes follows a "file" metaphor. When a class is encountered, the following steps are performed :

- Open a new class.
- Inherit from base classes.
- Add members to the class (functions, variables, constants, etc...)
- Close the class and emit object-oriented code.

As a class is constructed, a language module may need to keep track of a variety of data such as whether constructors or destructors have been given, are there any data members, have datatypes been renamed, and so on. It is not always a clear-cut process.

Language extensions

Providing additional support for object-oriented programming requires the use of the following Language extensions. These are additional methods that can be defined for the Language class.

`void cpp_open_class(char *name, char *rename, char *ctype, int strip);`

Opens a new class. `name` is the name of the class, `rename` is the renamed version of the class (or NULL if not renamed), `ctype` is the class type (`struct`, `class`, `union`), and `strip` is a flag indicating whether or not its safe to drop the leading type specifier (this is often unsafe for ANSI C).

`void cpp_inherit(char **baseclass, int mode = INHERIT_ALL);`

Inherits from base classes. `baseclass` is a NULL terminated array of class names corresponding to all of the base classes of an object. `mode` is an inheritance mode that is the

or'd value of INHERIT_FUNC , INHERIT_VAR, INHERIT_CONST, or INHERIT_ALL.

void cpp_member_func(char *name, char *iname, DataType *t, ParmList *l);

Creates a member function. name is the real name of the member, iname is the renamed version (NULL if not renamed), t is the return datatype, and l is the function parameter list.

void cpp_static_func(char *name, char *iname, DataType *t, ParmList *l);

Create a static member function. The calling conventions are the same as for cpp_member_func().

void cpp_variable(char *name, char *iname, DataType *t);

Creates a member variable. name is the real name of the member, iname is the renamed version (NULL is not renamed). t is the type of the member.

void cpp_static_var(char *name, char *iname, DataType *t);

Creates a static member variable. The calling convention is the same as for cpp_variable().

void cpp_declare_const(char *name, char *iname, DataType *type, char *value);

Creates a constant inside a C++ class. Normally this is an enum or member declared as const. name is the real name, iname is the renamed version (NULL if not renamed), type is the type of the constant, and value is a string containing the value.

void cpp_constructor(char *name, char *iname, ParmList *l);

Creates a constructor. name is the name of the constructor, iname is the renamed version, and l is the function parameter list. Normally, name is the same name as the class. If not, this may actually be a member function with no declared return type (assumed to be an int in C++).

void cpp_destructor(char *name, char *newname);

Creates a destructor. name is the real name of the destructor (usually the same name as the class), and newname is the renamed version (NULL if not renamed).

void cpp_close_class();

Closes the current class. Language modules should finish off code generation for a class once this has been called.

void cpp_cleanup();

Called after all C++ classes have been generated. Only used to provide some kind of global cleanup for all classes.

Hints

Special C++ code generation is not for the weak of heart. Most of SWIG's built in modules have been developed for well over a year and object oriented support has been in continual development. If writing a new language module, looking at the implementation for Python, Tcl, or Perl5

would be a good start.

Documentation Processing

The documentation system operates (for the most part), independently of the language modules. However, language modules are still responsible for generating a "usage" string describing how each function, variable, or constant is to be used in the target language.

Documentation entries

Each C/C++ declaration in an interface file gets assigned to a "Documentation Entry" that is described by the `DocEntry` object :

```
class DocEntry {
public:
  String      usage;           // Short description
  String      cinfo;           // Information about C interface (optional).
  String      text;            // Supporting text (optional)
};
```

The `usage` string is used to hold the calling sequence for the function. The `cinfo` field is used to provide additional information about the underlying C code. `text` is filled in with comment text.

The global variable `doc_entry` always contains the documentation entry for the current declaration being processed. Language modules can choose to update the documentation by referring to and modifying its fields.

Creating a usage string

To create a documentation usage string, language modules need to modify the 'usage' field of `doc_entry`. This can be done by creating a function like this :

```
// ----------------------------------------------------------------------
// char *TCL::usage_string(char *iname, DataType *t, ParmList *l),
//
// Generates a generic usage string for a Tcl function.
// ----------------------------------------------------------------------

char * TCL::usage_string(char *iname, DataType *, ParmList *l) {

  static String temp;
  Parm  *p;
  int   i, numopt,pcount;

  temp = "";
  temp << iname << " ";

  /* Now go through and print parameters */
  i = 0;
  pcount = l->nparms;
  numopt = l->numopt();
  p = l->get_first();
  while (p != 0) {
    if (!p->ignore) {
```

```
            if (i >= (pcount-numopt))
              temp << "?";

            /* If parameter has been named, use that.   Otherwise, just print a type  */

            if ((p->t->type != T_VOID) || (p->t->is_pointer)) {
              if (strlen(p->name) > 0) {
                temp << p->name;
              }
              else {
                temp << "{ " << p->t->print_type() << " }";
              }
            }
            if (i >= (pcount-numopt))
              temp << "?";
            temp << " ";
            i++;
          }
          p = l->get_next();
        }
        return temp;
      }
```

Now, within the function to create a wrapper function, include code such as the following :

```
// Fill in the documentation entry
doc_entry->usage << usage_string(iname,t,l);
```

To produce full documentation, each language module needs to fill in the documentation usage string for all declarations. Looking at existing SWIG modules can provide more information on how this should be implemented.

Writing a new documentation module

Writing a new documentation module is roughly the same idea as writing a new Language class. To do it, you need to implement a "Documentation Object" by inheriting from the following base class and defining all of the virtual methods :

```
class Documentation {
public:
  virtual void parse_args(int argc, char **argv) = 0;
  virtual void title(DocEntry *de) = 0;
  virtual void newsection(DocEntry *de, int sectnum) = 0;
  virtual void endsection() = 0;
  virtual void print_decl(DocEntry *de) = 0;
  virtual void print_text(DocEntry *de) = 0;
  virtual void separator() = 0;
  virtual void init(char *filename) = 0;
  virtual void close(void) = 0;
  virtual void style(char *name, char *value) = 0;
};
```

`void parse_args(int argc, char **argv);`

Parses command line options. Any special options you want to provide should be placed here.

`void title(DocEntry *de;`
Produces documentation output for a title.

`void newsection(DocEntry *de, int sectnum;`
Called whenever a new section is created. The documentation system is hierarchical in nature so each call to this function goes down one level in the hierarchy.

`void endsection();`
Ends the current section. Moves up one level in the documentation hierarchy.

`void print_decl(DocEntry *de);`
Creates documentation for a C declaration (function, variable, or constant).

`void print_text(DocEntry *de);`
Prints documentation that has been given with the `%text %{ %}` directive.

`void separator();`
Prints an optional separator between sections.

`void init(char *filename);`
Initializes the documentation system. This function is called after command line options have been parsed. `filename` is the file where documentation should be placed.

`void close(void);`
Closes the documentation file. All remaining output should be complete and files closed upon exit.

`void style(char *name, char *value);`
Called to set style parameters. This function is called by the `%style` and `%localstyle` directives. It is also called whenever style parameters are given after a section directive.

Using a new documentation module

Using a new documentation module requires a change to SWIG's main program. If you are writing your own main() program, you can use a new documentation module as follows :

```
#include <swig.h>
#include "swigtcl.h"                    // Language specific header
#include "mydoc.h"                      // New Documentation module

extern int SWIG_main(int, char **, Language *, Documentation *);
int main(int argc, char **argv) {

       TCL *l = new TCL;                // Create a new Language object
       MyDoc *d = new MyDoc;            // New documentation object
       init_args(argc, argv);           // Initialize args
       return SWIG_main(argc, argv, l, d);
}
```

Where to go for more information

To find out more about documentation modules, look at some of the existing SWIG modules contained in the `SWIG1.1/SWIG` directory. The ASCII and HTML modules are good starting points for finding more information.

The Future of SWIG

SWIG's C++ API is the most rapidly evolving portion of SWIG. While interface-file compatibility will be maintained as much as possible in future releases, the internal structure of SWIG is likely to change significantly in the future. This will possibly have many ramifications on the construction of language modules. Here are a few significant changes that are coming :

1. A complete reorganization of the SWIG type system. Datatypes will be represented in a more flexible manner that provide full support for arrays, function pointers, classes, structures, and types possibly coming from other languages (such as Fortran).
2. Objectification of functions, variables, constants, classes, etc... Currently many of SWIG's functions take multiple arguments such as functions being described by name, return datatype, and parameters. These attributes will be consolidated into a single "Function" object, "Variable" object, "Constant" object, and so forth.
3. A complete rewrite of the SWIG parser. This will be closely tied to the change in datatype representation among other things.
4. Increased reliance on typemaps. Many of SWIG's older modules do not rely on typemaps, but this is likely to change. Typemaps provide a more concise implementation and are easier to maintain. Modules written for 1.1 that adopt typemaps now will be much easier to integrate into future releases.
5. A possible reorganization of object-oriented code generation. The layered approach will probably remain in place however.
6. Better support for multiple-files (exporting, importing, etc...)

In planning for the future, much of a language's functionality can be described in terms of typemaps. Sticking to this approach will make it significantly easier to move to new releases. I anticipate that there will be few drastic changes to the Language module presented in this section (other than changes to many of the calling mechanisms).